Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерних наук і технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти – бакалавр
за освітньо-професійною програмою
«Комп'ютерні науки»
зі спеціальності
122 – Комп'ютерні науки

тема роботи:

«Розробка веб-інтерфейсу та серверної частини системи організації персонального простору»

Виконала ст. гр. КН-22-2 _____ Солодаренко Н. О.

Керівник _____ Рубан С. А.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2026

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет: *інформаційних технологій*

Кафедра: *автоматизації, комп'ютерних наук і технологій*

Ступінь вищої освіти: *Бакалавр*

Спеціальність: *122 – Комп'ютерні науки*

ЗАТВЕРДЖУЮ

Зав. кафедрою: *к.т.н. Рубан С.А.*

_____

« *26* » *січня* 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу бакалавра

студентові групи КН-22-2 *Солодаренко Наталії Олександрівні*

1. Тема кваліфікаційної роботи: *«Розробка веб-інтерфейсу та серверної частини системи організації персонального простору»*

затверджено наказом по університету № *173с* від *30.03.2026 р.*

2. Термін здачі кваліфікаційної роботи: *10.06.2026 р.*

3. Склад кваліфікаційної роботи: *Пояснювальна записка обсягом 75с., додатки, презентація у Microsoft PowerPoint (12 слайдів) в електронному та друкованому вигляді*

4. Консультанти кваліфікаційної роботи:

Розділ 1-2 доц. Рубан С. А.

Нормоконтроль доц. Маринич І. А.

5. Календарний план:

№	Етапи роботи	Термін виконання
1	*Вступ*	*01.04.26*
2	*Розділ 1*	*05.04.26*
3	*Розділ 2*	*01.05.26*
4	*Висновки*	*25.05.26*
5	*Оформлення кваліфікаційної роботи*	*28.05.26*
6	*Підготовка презентації та графічного матеріалу*	*20.05.26*
7	*Підготовка доповіді до захисту*	*10.06.26*

6. Дата видачі завдання: *28.01.2026р.*

Керівник _____ / Рубан С. А./

7. Запевнення: *Я, Солодаренко Наталія Олександрівна, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*

Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.

Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.

Студентка _____ / Солодаренко Н. О./

АНОТАЦІЯ

Солодаренко Н. О. Розробка веб-інтерфейсу та серверної частини системи організації персонального простору: кваліфікаційна робота бакалавра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2026. 75 с.

Актуальність теми полягає в тому, що стрімко зростає потреба в гнучких інструментах для організації персонального простору, з метою підвищити власну продуктивність. Наразі платформи із високим рівнем кастомізації інтерфейсу потребують дослідження нових архітектурних підходів, оптимізації клієнт-серверної взаємодії, щоб забезпечити високу швидкодію, та масштабованості системи.

Метою роботи є розробка гнучкої веб-платформи для організації персонального простору з можливістю редагування користувацького інтерфейсу за допомогою віджетів та впровадження механізмів оптимізації клієнтської частини.

У першому розділі було розглянуто існуючі SaaS-платформи для організації даних та виявлено їхні архітектурні обмеження, також проведено порівняльний аналіз між різними моделями баз даних і проаналізовано архітектурні стилі організації API (REST та GraphQL). В результаті було обрано такий технологічний стек: .NET, ORM Entity Framework Core, Hot Chocolate та React.

У другому розділі розглядаються проєктування та технічна реалізація веб-платформи організації персонального простору. Також описуються впровадження гібридної структури збереження даних, реалізація атомарних операцій на основі патерну Unit of Work, оптимізація фронтенду та реалізація відкладеного збереження.

Ключові слова:

ОРГАНІЗАЦІЯ ПРОСТОРУ, ВЕБ-ПЛАТФОРМА, КОНСТРУКТОР ІНТЕРФЕЙСУ, ГІБРИДНА МОДЕЛЬ ДАНИХ, .NET, GRAPHQL, HOT CHOCOLATE, REACT, FLUENT API, UNIT OF WORK

ЗМІСТ

ВСТУП

Ефективна організація особистого простору є критично важливим завданням в житті будь-якої людини, особливо зараз, коли цифрової інформації в нашому світі з кожним днем стає тільки більше.

Існуючі системи керуванням інформацією часто не можуть задовольнити потреби користувачів через багато факторів: жорстко задану структуру інтерфейсу, високий поріг входу, недоступність важливого функціоналу для безплатних тарифів.

Актуальність цієї роботи полягає у розробці гнучкої платформи (конструктору простору), яка може дозволити користувачам самостійно організовувати власний особистий простір за допомогою інтерактивних віджетів.

Метою дослідження є проєктування та програмна реалізація веб-платформи для організації персонального простору із використанням сучасного стеку технологій, що забезпечить гнучке керування віджетами, високу швидкодію та цілісність даних.

Для досягнення цієї мети було вирішено такі завдання:

– аналіз існуючих SaaS-платформ для організації даних та виявлення їхніх архітектурних обмежень та недоліків;

– дослідження підходів до збереження даних;

– аналіз архітектурних стилів організації API для систем із динамічною структурою інтерфейсу;

– проєктування структури бази даних та налаштування зв’язків між сутностями;

– розробка серверної частини системи з впровадженням патерну Unit of Work;

– реалізація гнучкого API для каскадного отримання даних та автентифікації користувачів;

– розробка клієнтського інтерфейсу на базі динамічної адаптивної сітки з відкладеним збереженням змін та локалізацією.

Результатом дипломного проєкту є повністю працездатна, оптимізована веб-платформа, яка може бути впроваджена як індивідуальний інструмент підвищення продуктивності та консолідації розрізнених даних.

Практична цінність розроблених рішень полягає у:

– архітектурній масштабованості сховища даних, оскільки впроваджений гібридний підхід дозволяє розширювати функціонал платформи та додавати нові типи інформаційних віджетів без деструктивних міграцій та зміни схеми бази даних на сервері;

– підвищенні надійності системного програмного забезпечення, оскільки розроблений проміжний обробник на базі патерну Unit of Work гарантує атомарність модифікації даних та унеможливлює розсинхронізацію між СУБД та файловою системою під час роботи з медіафайлами;

– оптимізації мережевого навантаження, оскільки реалізований на клієнтській частині механізм пакетного збереження чернеток мінімізує кількість запитів до API, заощаджує ресурси сервера та забезпечує миттєвий візуальний відгук користувацького інтерфейсу.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ МЕТОДІВ ТА ІНСТРУМЕНТІВ РОЗРОБКИ СИСТЕМ ОРГАНІЗАЦІЇ ПЕРСОНАЛЬНОГО ПРОСТОРУ

1.1 Аналіз сучасних платформ організації даних та їхніх архітектурних особливостей

Сьогодні для організації особистих справ та збереження інформації люди використовують абсолютно різні платформи. Хтось намагається збирати все в одному місці за допомогою гнучких конструкторів, хтось керує завданнями через прості візуальні дошки, а для обліку грошей чи аналітики взагалі доводиться завантажувати окремі програми. Щоб зрозуміти, як правильніше побудувати архітектуру нової системи, ми розглянемо три найпопулярніші платформи, кожна з яких представляє свій підхід до роботи з даними та інтерфейсом.

1.1.1 Аналіз платформи «Notion»

Першим об'єктом дослідження є платформа «Notion», розроблена американською компанією Notion Labs Inc.. На сьогоднішній день цей інструмент позиціонується як універсальна робоча область, що базується на унікальній блоковій архітектурі. Користувачі мають можливість конструювати персональний інформаційний простір, комбінуючи текстові блоки, мультимедійні елементи, інтегрований програмний код та складні реляційні бази даних.

Функціональні можливості системи включають розширений інструментарій для створення Wiki-сторінок, ієрархічних нотаток та канбан-дошок для управління проєктами. Особливої уваги заслуговує гнучкість налаштування баз даних, які можуть відображатися у вигляді таблиць, календарів або галерей, що дозволяє адаптувати платформу під будь-яку предметну область — від особистого щоденника до корпоративної системи управління знаннями.

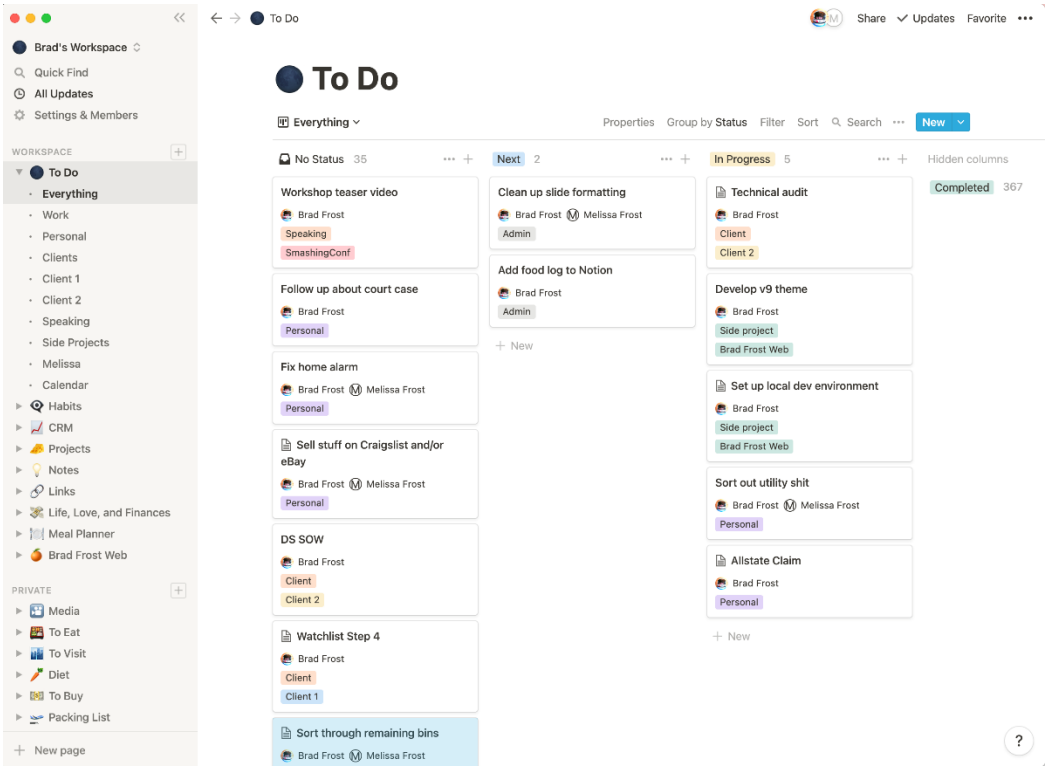

Рисунок 1.1 – Платформа «Notion»

Перевагою Notion є його безпрецедентна адаптивність. Завдяки відсутності жорстко заданої структури, користувачі можуть теоретично організувати облік будь-яких даних в межах одного інтерфейсу, уникаючи потреби у використанні сторонніх сервісів для зберігання документів чи списків завдань.

Проте, під час детального аналізу було виявлено низку суттєвих недоліків. Головним бар'єром для масових користувачів є надто високий поріг входу: для створення ефективного робочого простору необхідно витратити значну кількість часу на вивчення логіки роботи блоків та налаштування зв'язків між базами. Платформа функціонує як складний конструктор, але не пропонує готових, інтуїтивно зрозумілих рішень для повсякденного вжитку. Зокрема, у Notion відсутні інтегровані модулі глибокої аналітики (наприклад, автоматичні графіки фінансової динаміки чи універсальні звіти), що змушує користувачів створювати їх вручну за допомогою складних формул. Окрім цього, критичними мінусами є низька продуктивність при роботі з великими обсягами даних та обмежена

функціональність в офлайн-режимі, що знижує надійність системи для оперативного управління персональним простором.

1.1.2 Аналіз платформи «Trello»

Наступним об'єктом аналізу є платформа «Trello», яка належить австралійській компанії Atlassian. На відміну від універсального Notion, Trello є вузькоспеціалізованим інструментом, що базується на методології Kanban. Основна концепція системи побудована навколо візуальних дошок, списків та карток, що дозволяє користувачам ефективно керувати потоками завдань у реальному часі.

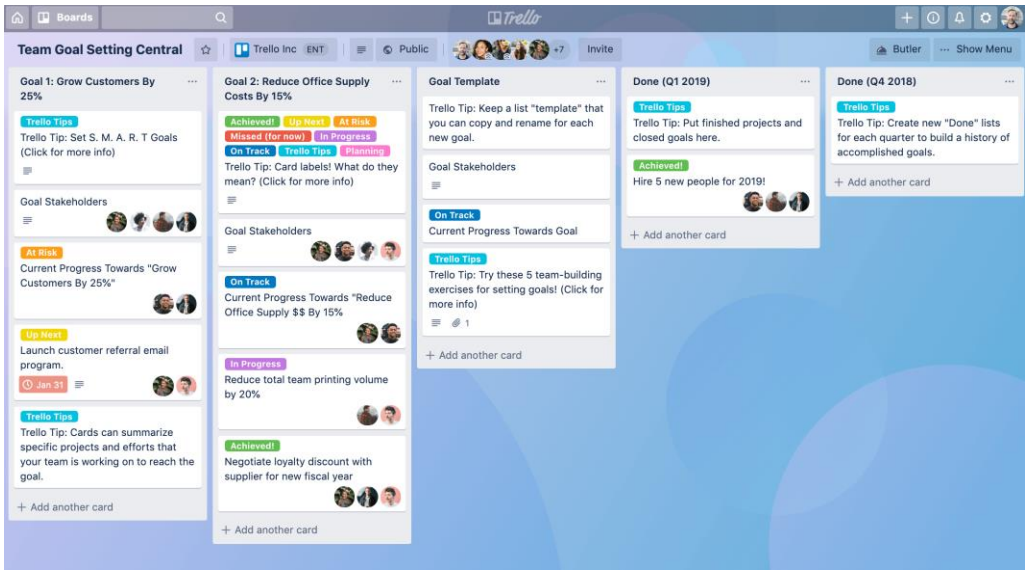

Рисунок 1.2 – Платформа «Trello»

Функціональні можливості платформи зосереджені на операційному управлінні: встановлення дедлайнів, призначення відповідальних (у разі командної роботи), створення чек-листів та додавання вкладень до конкретних карток. Система також пропонує автоматизацію рутинних дій за допомогою вбудованого інструменту Butler та широкі можливості інтеграції зі сторонніми сервісами через систему Power-Ups.

Головною перевагою Trello є максимальна простота використання та

мінімалістичний, інтуїтивно зрозумілий інтерфейс. Це забезпечує низький поріг входу — користувачі можуть почати роботу без попереднього навчання, що робить платформу ідеальним рішенням для швидкого планування справ.

Проте, як показав аналіз предметної області, критичним недоліком системи є її надмірна функціональна обмеженість (жорстка спеціалізація). Платформа позбавлена інструментів для ведення глибоких ієрархічних баз знань, трекінгу складних аналітичних даних або фінансового обліку. Trello виступає класичним прикладом інструменту, що провокує фрагментацію цифрового простору: користувачі змушені переходити до інших додатків для ведення нотаток чи обліку витрат, що веде до втрати контексту та зниження загальної продуктивності. Така ізольованість даних унеможливлює побудову цілісної картини особистого прогресу, де завдання були б безпосередньо пов'язані з ресурсами та знаннями.

1.1.3 Аналіз платформи «Finmap»

Завершальним об’єктом порівняльного аналізу є платформа «Finmap», розроблена однойменною українською компанією. Даний сервіс є вузькоспрямованим інструментом для фінансового обліку, який, хоча й орієнтований на малий та середній бізнес, часто адаптується користувачами для управління персональними фінансами.

Функціональні можливості системи включають глибоку та професійну аналітику грошових потоків, деталізовані звіти за категоріями, мультивалютний облік та можливість автоматичної синхронізації з банківськими виписками. Чітка структура відстеження доходів і витрат дозволяє користувачу отримати прозору картину свого фінансового стану в короткостроковій та довгостроковій перспективах.

Сильною стороною Finmap є висока якість візуалізації фінансових даних та автоматизація складних звітів, що робить його одним із лідерів у своїй ніші на вітчизняному ринку. Професійний підхід до структурування транзакцій забезпечує високу точність обліку, яку важко реалізувати в універсальних конструкторах типу

Notion.

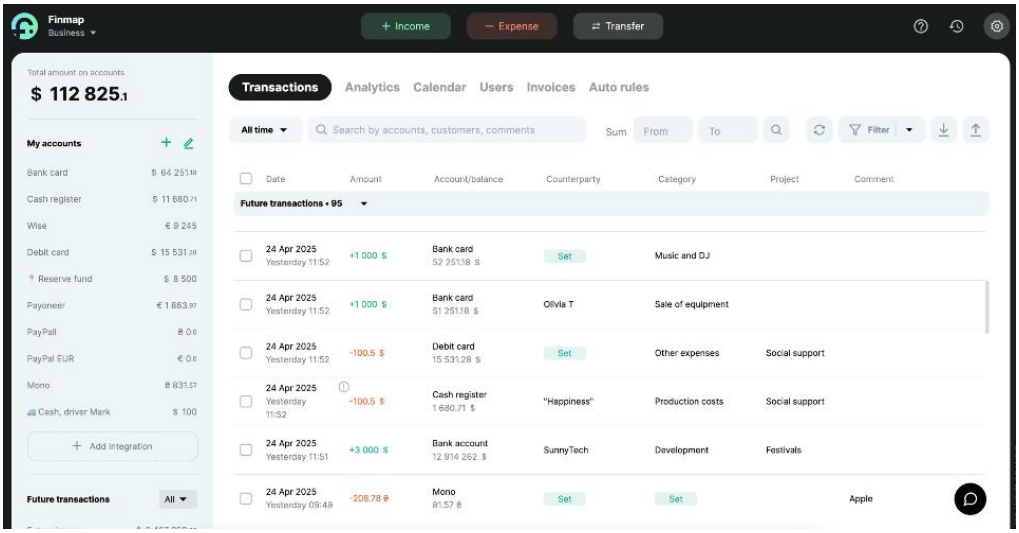

Рисунок 1.3 – Платформа «Finmap»

Проте, з точки зору концепції цілісного персонального простору, сервіс має ключовий недолік — повну ізоляцію даних. Платформа функціонує виключно в межах однієї предметної області, що унеможливлює створення гнучких структур для трекінгу освітніх проєктів, особистих навичок чи контенту. Головною проблемою для користувачів є неможливість встановити кореляцію між фінансовими показниками та особистими цілями чи звичками. Користувачі не отримують відповіді на питання, як інвестиції в навчання чи зміна щоденних звичок впливають на його загальну фінансову динаміку, оскільки система залишається закритою.

1.2 Дослідження підходів до збереження даних

Для побудови інформаційної системи, яка дозволяє користувачу динамічно змінювати структуру свого робочого простору, ключовим архітектурним викликом є вибір моделі організації бази даних. Ефективність платформи, її масштабованість та швидкість відповіді на запити напряму залежать від того, як саме сервер

обробляє та зберігає конфігурації користувацьких віджетів. З огляду на це, необхідно проаналізувати три основні підходи до моделювання даних, щоб знайти оптимальний баланс між гнучкістю та надійністю.

Класична реляційна модель даних (SQL) будується на основі суворої структури у вигляді зв'язаних таблиць із фіксованим набором колонок. Для стандартних інформаційних систем це ідеальний вибір, оскільки він гарантує цілісність даних, підтримує транзакції та дозволяє будувати чіткі зв'язки між сутностями за допомогою зовнішніх ключів. Проте, якщо спробувати реалізувати на чистому SQL систему, де користувачі мають мати змогу додати на сторінку будь-який тип віджета із власними унікальними налаштуваннями, реляційний підхід створює серйозні проблеми. При розробці доведеться або постійно змінювати схему бази даних при додаванні нових типів віджетів, або використовувати складний патерн Entity-Attribute-Value, який через нескінченні операції об'єднання таблиць критично знижує швидкість роботи сервера.

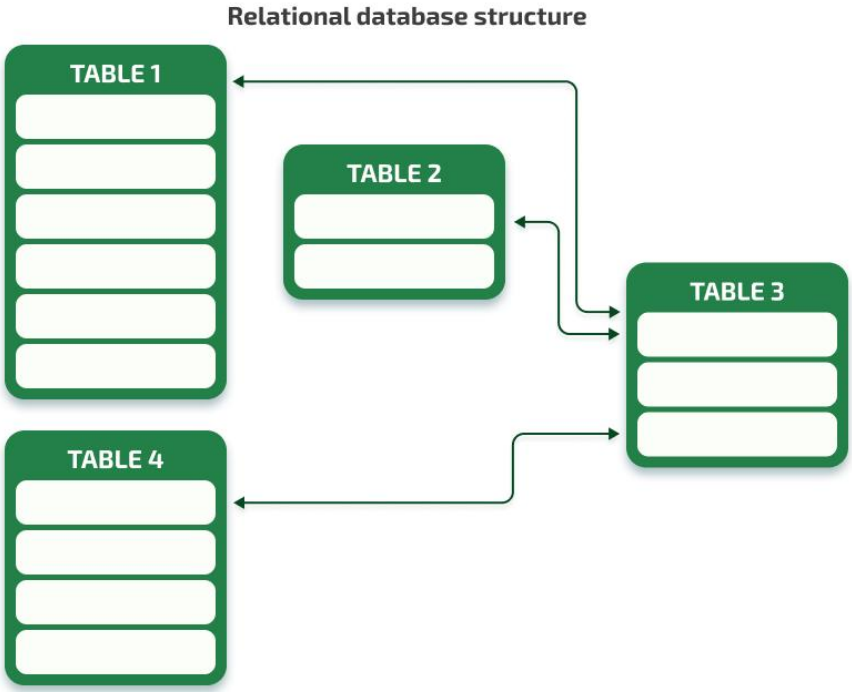

Рисунок 1.4 – Реляційна модель даних

Альтернативна документо-орієнтована модель даних (NoSQL) пропонує зберігати інформацію у вигляді гнучких документів, які не мають фіксованої схеми.

Це розв'язує руки при розробці конструкторів інтерфейсу, адже кожен віджет чи конфігурація сітки може бути записана як окремий JSON-об'єкт із будь-яким набором полів, а клієнтська частина може миттєво зберігати свій стейт без додаткових перевірок на сервері. З іншого боку, повний перехід на NoSQL-рішення руйнує базові реляційні зв'язки, які необхідні для стабільної роботи системи. Такі сутності, як акаунти користувачів, права доступу, логіка JWT-авторизації та ієрархія розділів потребують суворого контролю на рівні бази даних, і відсутність жорстких конструкцій у NoSQL суттєво підвищує ризик появи сміттєвих даних або порушення безпеки.

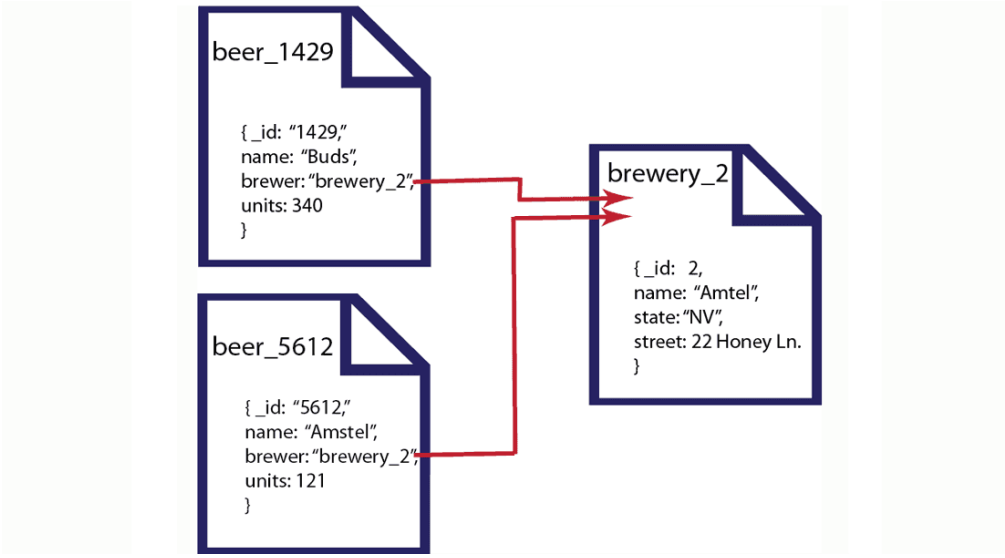

Рисунок 1.5 – Документо-орієнтована модель даних

Сучасним компромісом для вирішення цієї дилеми є гібридна модель даних, що поєднує переваги обох світів в межах однієї СУБД (наприклад, через підтримку типу JSONB у PostgreSQL). Цей підхід дозволяє розділити інформаційну структуру платформи на дві логічні частини: суворо структуровані системні дані зберігаються у класичних таблицях із чіткими зв'язками та каскадним видаленням, а динамічні налаштування віджетів та координати сітки інтерфейсу записуються в JSON-поля прямо всередині цих таблиць.

Для архітектури нашого проєкту було обрано саме гібридну модель збереження даних. Таке рішення дозволяє повністю уникнути труднощів з

важкими запитами, дає змогу зберігати складні конфігурації фронтенд-сітки в один рядок бази даних, і при цьому залишає надійний контроль над безпекою, користувачами та зв'язками між робочими просторами.

1.3 Порівняльний аналіз REST API та GraphQL для систем із динамічним інтерфейсом

Організація ефективної взаємодії між клієнтською та серверною частинами є наступним критично важливим етапом проєктування систем із високим рівнем кастомізації. Оскільки інтерфейс нашої платформи складається з динамічного набору віджетів, структура даних, яку запитує фронтенд, постійно змінюється залежно від налаштувань конкретних користувачів. Для реалізації цього обміну необхідно проаналізувати два провідні архітектурні підходи — класичний стиль REST API та сучасний протокол GraphQL, щоб визначити їхні переваги та обмеження в межах розроблюваної системи.

Традиційний архітектурний стиль REST API базується на концепції фіксованих кінцевих точок (endpoints), де кожен ресурс має свою чітку URL-адресу, а сервер завжди повертає строго визначену структуру JSON. У системах із динамічним інтерфейсом такий підхід породжує дві серйозні проблеми: надлишкове завантаження даних (Overfetching) та недостатність даних у запиті (Underfetching). Наприклад, для рендерингу однієї сторінки конструктора простору клієнту довелося б або робити серію послідовних запитів до різних кінцевих точок (щоб зібрати дані робочого простору, конфігурацію сітки та вміст усіх віджетів), що створює велике навантаження на мережу, або сервер мав би повертати один гігантський масив інформації, більша частина якого може бути непотрібною в цей конкретний момент.

Альтернативна технологія GraphQL пропонує принципово інший підхід, де клієнт сам визначає структуру відповіді за допомогою спеціальної мови запитів (Queries та Mutations), взаємодіючи лише з однією кінцевою точкою сервера. Це

ідеально підходить для концепції гнучкого конструктора, оскільки дозволяє фронтенду на React одним каскадним запитом дістати всю необхідну ієрархію сторінки — від назви розділу до точних координат розташування віджетів та їхнього вмісту. Використання GraphQL повністю нівелює проблему надлишкового завантаження даних, оптимізує споживання трафіку та дозволяє серверній частині на .NET за допомогою інструменту Hot Chocolate гнучко транслювати клієнтські запити прямо в оптимізовані вибірки з бази даних.

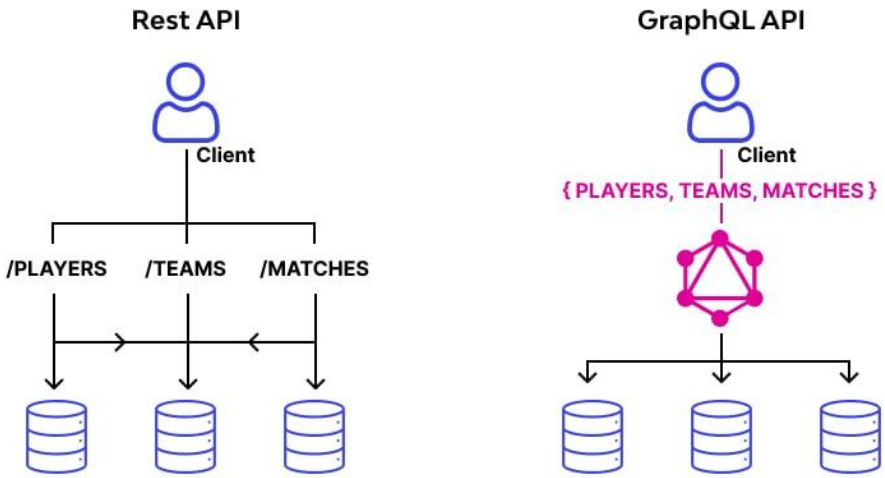

Рисунок 1.6 – Протоколи передачі даних REST API та GraphQL

Проте, незважаючи на очевидні переваги GraphQL для роботи зі структурованим контентом, цей протокол має серйозне архітектурне обмеження, пов'язане з передачею бінарних даних. Спроба передавати важкі медіафайли, зображення чи документи через GraphQL-мутації у форматі Base64 або за допомогою складних механізмів Multipart Upload призводить до надмірного навантаження на процесор, збільшує обсяг трафіку приблизно втричі через особливості кодування та ускладнює логіку серверові.

Для реалізації нашої платформи було прийнято рішення впровадити гібридний підхід, поєднавши обидва інструменти для вирішення конкретних архітектурних завдань. Основна бізнес-логіка, яка відповідає за конфігурацію простору,

управління віджетами, зміну стейту та автентифікацію користувачів, повністю перенесена на GraphQL, що забезпечує максимальну гнучкість та швидкість роботи інтерфейсу. Натомість для завантаження, збереження та обробки медіафайлів (зображень користувачів, обкладинок просторів тощо) реалізовано класичний REST API контролер, який дозволяє ефективно стрімити бінарні дані безпосередньо у сховище серверної частини без зайвих архітектурних накладних витрат.

1.4 Дослідження архітектурних концепцій побудови веб-інтерфейсів

Під час проєктування клієнтської частини інформаційної системи ключовим архітектурним викликом є вибір концепції відображення сторінок та розподілу обчислювального навантаження між браузером користувачів та сервером. Сучасна веб розробка пропонує два фундаментально різних підходи до вирішення цього завдання: класичну модель багатосторінкових застосунків (MPA — Multi-Page Application) та концепцію односторінкових застосунків (SPA — Single-Page Application). Оскільки інтерфейс нашої платформи має функціонувати як гнучкий конструктор із високим рівнем інтерактивності, необхідно детально проаналізувати ці моделі для забезпечення максимальної швидкодії системи.

Класичний підхід багатосторінкових застосунків (MPA) базується на парадигмі, де кожна дія користувачів (наприклад, перехід на іншу сторінку чи відправка форми) викликає повноцінний запит до сервера, який заново генерує HTML-код сторінки та повертає його в браузер. Для інформаційних систем із жорсткою та статичною структурою цей підхід є традиційним і надійним, проте для конструктора особистого простору він створює критичні обмеження. Оскільки наша платформа передбачає постійну динамічну взаємодію користувачів з інтерфейсом (перетягування віджетів, зміну їхніх розмірів, редагування вмісту в реальному часі), будь-яке повне перезавантаження сторінки руйнувало б плавність інтерфейсу, переривало б стан компонентів та створювало б колосальне

надлишкове навантаження на сервер.

Альтернативна архітектурна модель односторінкових застосунків (SPA) пропонує завантажувати базову логіку, стилі та скрипти інтерфейсу в браузер користувачів лише один раз під час першого звернення до вебсайту. Подальша взаємодія із сервером відбувається асинхронно у фоновому режимі, а інтерфейс оновлюється динамічно за допомогою JavaScript без перезавантаження всієї сторінки. Такий підхід дозволяє повністю перенести обчислення, пов'язані з позиціонуванням віджетів на сітці, локальним кешуванням змін та валідацією користувацьких налаштувань, на сторону клієнта, реалізуючи концепцію «розумного клієнта» (Smart Client). Сервер при цьому звільняється від рутини рендерингу графічного інтерфейсу і функціонує виключно як ізольоване, високоефективне API для зчитування та збереження сирих даних, що суттєво підвищує загальну масштабованість системи під великими навантаженнями.

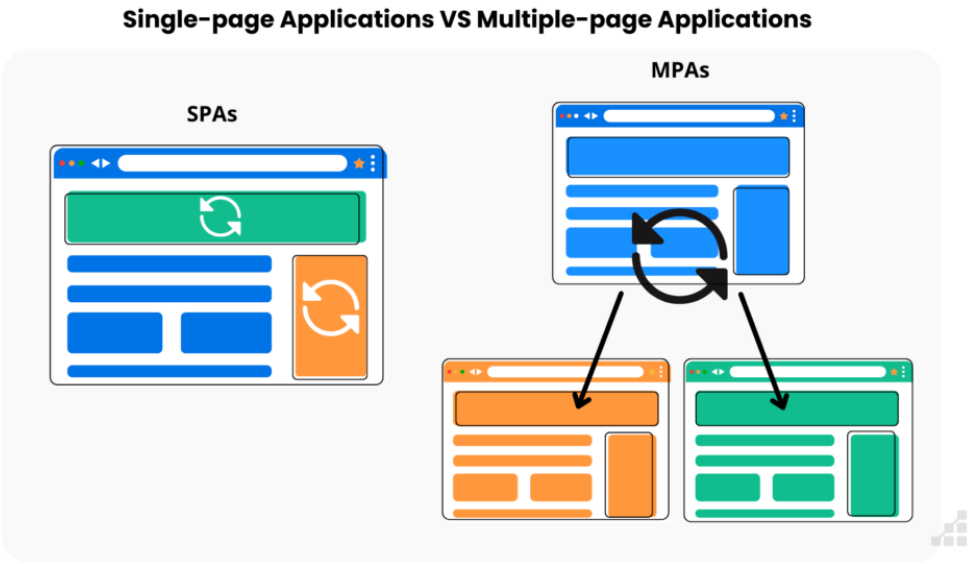

Рисунок 1.7 – Концепції побудови вебінтерфейсів MPA та SPA

Для архітектури нашої платформи було обрано саме концепцію односторінкових застосунків (SPA). Це архітектурне рішення забезпечує миттєвий відгук інтерфейсу на дії користувачів, дозволяє безшовно інтегрувати складні фронтенд-бібліотеки для керування динамічними сітками та мінімізує обсяг

трафіку між клієнтом і сервером, обмінюючись виключно компактними JSON-пакетами.

1.5 Обґрунтування вибору технологічного стеку для реалізації веб-платформи

Фінальним етапом теоретичного дослідження є вибір інструментальних засобів та технологій, які дозволять практично реалізувати спроєктовану гібридну архітектуру нашої системи. Для забезпечення високої швидкодії, безпеки даних та гнучкості інтерфейсу необхідно підібрати стекові рішення, що мають високий рівень синергії між собою, активну підтримку спільноти та оптимізовані під сучасні хмарні розгортання.

1.5.1 Вибір серверної платформи та ORM-системи

Екосистема .NET (зокрема фреймворк ASP.NET Core) була обрана як базова платформа для розробки серверної частини нашої вебплатформи. Ця технологія забезпечує високий рівень продуктивності та кросплатформенності, що дозволяє розгортати сервер у будь-яких контейнеризованих середовищах. Мова програмування C# надає строгу типізацію, що мінімізує кількість помилок на етапі компіляції, а вбудовані механізми ін'єкції залежностей (Dependency Injection) та асинхронного програмування дозволяють побудувати масштабовану та стійку до високих навантажень бізнес-логіку.

Для взаємодії з базою даних та реалізації об'єктно-реляційного відображення обрано фреймворк Entity Framework Core (EF Core). Головною перевагою EF Core є підтримка підходу Code-First, що дозволяє описувати структуру бази даних за допомогою C# класів, а також потужний інструмент Fluent API для гнучкого налаштування каскадних зв'язків та обмежень цілісності. Як систему управління базами даних (СУБД) обрано PostgreSQL, оскільки вона є безкоштовним рішенням з відкритим кодом, має високу надійність і, що найважливіше для нашого проєкту, пропонує найкращу на ринку підтримку бінарного типу даних JSONB для

збереження динамічних конфігурацій віджетів.

1.5.2 Вибір архітектурного стилю API та інструментів організації запитів

Побудова GraphQL API на серверній стороні реалізована за допомогою екосистеми Hot Chocolate. Цей GraphQL-сервер для платформи .NET дозволяє створювати схеми даних за принципом Code-First, автоматично транслюючи C# моделі у GraphQL типи. Hot Chocolate забезпечує високу швидкість парсингу запитів та безшовно інтегрується з EF Core, дозволяючи автоматично оптимізувати SQL-запити до бази даних за допомогою механізмів проекцій (з бази дістаються лише ті колонки, які клієнт реально запросив у query).

Класичний архітектурний стиль REST API також інтегровано в систему для вирішення специфічного кола завдань, де GraphQL демонструє низьку ефективність. Зокрема, для завантаження, збереження та обробки медіафайлів реалізовано класичний REST контролер. Це дозволяє ефективно стрімити бінарні дані безпосередньо у сховище серверної частини без зайвих архітектурних накладних витрат та кодування файлів у Base64-рядки.

1.5.3 Вибір інструментів фронтенд-розробки

Клієнтська частина вебплатформи проєктується на базі бібліотеки React, яка є лідером у сфері створення динамічних односторінкових застосунків (SPA). Використання компонентного підходу в React дозволяє створити ізольовані, перевикористовувані модулі для кожного типу віджета, а концепція Virtual DOM забезпечує миттєве оновлення інтерфейсу без повного перезавантаження сторінки.

Для інтерактивного керування розташуванням елементів обрано бібліотеку react-grid-layout, яка надає готову адаптивну сітку з підтримкою технології Drag-and-Drop, що є критично важливим для створення інтуїтивного конструктора. Зв'язок із сервером на фронтенді забезпечується за допомогою Apollo Client, який бере на себе всю логіку кешування GraphQL-запитів, оптимізуючи кількість

звернень до мережі та спрощуючи управління глобальним стейтом застосунку.

Висновки до розділу:

У першому розділі проведено аналіз сучасних підходів до побудови вебсистем організації персонального простору. На основі дослідження існуючих SaaS-платформ (Notion, Trello, Finmap) було виявлено їхні ключові обмеження, зокрема проблему функціональної ізольованості даних, жорсткості інтерфейсів та високого порогу входу.

Досліджено моделі збереження даних та обґрунтовано доцільність використання гібридного підходу в межах СУБД PostgreSQL, що дозволяє поєднати сувору реляційну цілісність системних сутностей із гнучкістю документо-орієнтованого збереження конфігурацій віджетів. Здійснено порівняльний аналіз архітектурних стилів API, за результатом якого обрано комбіновану схему взаємодії: використання протоколу GraphQL для динамічної бізнес-логіки та класичного REST API для ефективної передачі бінарних медіафайлів. В кінці обґрунтовано вибір сучасного та продуктивного технологічного стеку, що базується на платформі .NET для серверної частини та бібліотеці React для розробки інтерактивного вебінтерфейсу.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА ТЕХНІЧНА РЕАЛІЗАЦІЯ СИСТЕМИ ОРГАНІЗАЦІЇ ПЕРСОНАЛЬНОГО ПРОСТОРУ

2.1 Розробка функціональних вимог та модульної архітектури платформи

Для веб-платформи організації персональної інформації ключовим завданням є надання інструментарію для безшовного керування простором. На основі аналізу предметної області та потреб потенційних користувачів було сформовано та деталізовано перелік обов'язкових функціональних вимог, які розділені за відповідними підсистемами:

1. Підсистема керування обліковими записами та безпекою:

– забезпечення надійної реєстрації нових користувачів із валідацією введених даних на стороні клієнта та сервера;

– автентифікація користувачів із використанням безпечного механізму JWT-токенів (JSON Web Tokens);

– реалізація системи розмежування доступу Row-Level Security, що унеможливлює доступ сторонніх осіб до чужих робочих просторів на рівні виконання GraphQL-запитів.

2. Підсистема управління структурою простору (простори та розділи):

– можливість створення кількох ізольованих робочих просторів (Workspaces) для розділення різних сфер життя (наприклад, «Робота», «Навчання», «Особисте»);

– реалізація ієрархічної структури розділів всередині простору за допомогою рекурсивних розділів, що дозволяє створювати необмежену вкладеність підрозділів;

– можливість динамічного перейменування, видалення та зміни обкладинок для будь-якого розділу з автоматичним каскадним оновленням

або видаленням вкладених елементів у базі даних.

3. Підсистема інтерактивного конструктора віджетів:

– надання користувачам гнучкої адаптивної сітки для візуального розташування елементів інтерфейсу за допомогою технології перетягування (Drag-and-Drop);

– підтримка поліморфної фабрики віджетів, що дозволяє динамічно додавати на сторінку блоки різних типів: текст, розділювачі, іконки, посилання, зображення, таблиці, канбан-дошки та графіки;

– реалізація механізму відкладеного збереження, який накопичує всі локальні зміни на сітці та вмісту віджетів у стейті фронтенду і відправляє їх на сервер одним масованим запитом лише при збереженні користувачами змін.

4. Підсистема управління медіаконтентом та локалізації:

– завантаження та збереження зображень для аватарів користувачів, обкладинок робочих просторів та вмісту медіа-віджетів через ізольовані REST-контролери;

– динамічне перемикання мови інтерфейсу системи (українська та англійська) без повного перезавантаження сторінки застосунку з автоматичною передачею локалізації в заголовках запитів до API.

Окрім функціональних вимог, до системи було висунуто низку нефункціональних вимог, які визначають якість та архітектурні обмеження розробки. Серед них:

– час відповіді GraphQL-сервера на запити читання конфігурації сторінки не повинен перевищувати 200 мс;

– інтерфейс платформи має бути адаптивним під різні роздільні здатності моніторів;

– архітектура системи повинна бути слабкопов'язаною для забезпечення простого додавання нових типів віджетів у майбутньому без модифікації існуючого ядра бекенду.

На рис. 2.1 представлено розроблену Use Case діаграму нашої вебплатформи. Основним актором системи виступає авторизований користувач, який безпосередньо ініціює процеси створення простору, взаємодіє з віджетами, керує ієрархією розділів та ініціює масове збереження змін. Неавторизований користувач має доступ виключно до реєстрації та входу в систему.

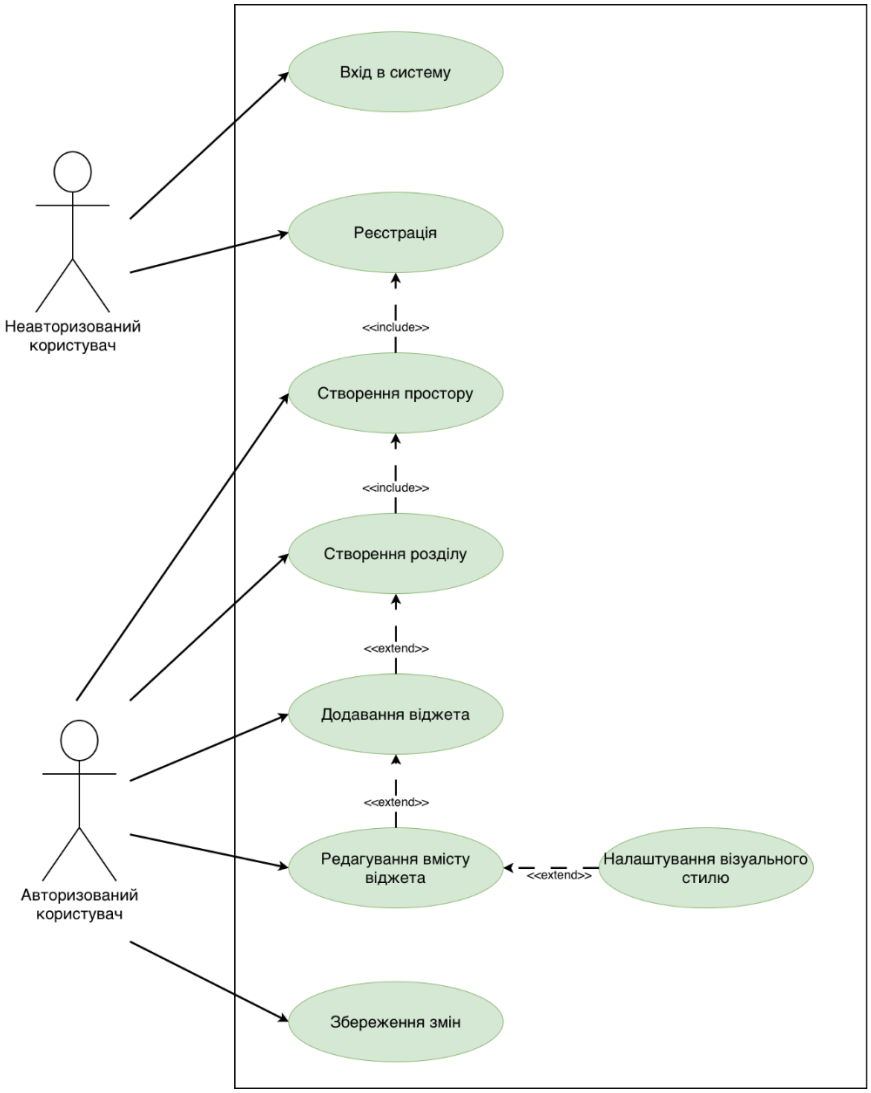

Рисунок 2.1 – Use Case діаграма

Для успішного виконання всіх сформульованих вимог було спроєктовано багаторівневу модульну архітектуру вебплатформи, яка базується на принципі розподілу відповідальності. Серверна архітектура на .NET побудована у вигляді набору слабкопов'язаних сервісів, замкнених на єдиному GraphQL. Модуль Identity ізольовано обробляє логіку користувачів та токенів. Модуль Workspace керує

релеційними зв'язками просторів та розділів. Модуль Widget відповідає за збереження серіалізованих JSON-конфігурацій віджетів у базі даних PostgreSQL. Модуль REST функціонує як REST-інфраструктура для стрімінгу бінарних даних.

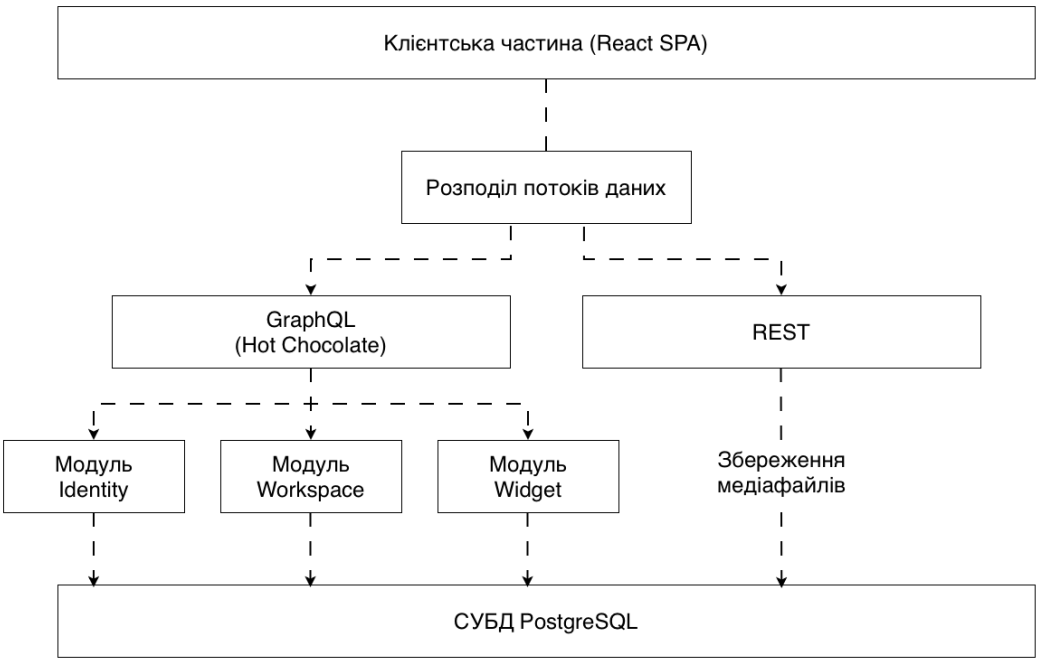

Рисунок 2.2 – Схема модульної архітектури вебплатформи

Клієнтська архітектура застосунку на React дзеркально відображає модульний підхід сервера, розділяючи фронтенд на три чіткі рівні: рівень представлення на базі 24-стовпчикової адаптивної сітки react-grid-layout, рівень управління станом, що реалізує логіку чернеток для збереження працездатності в умовах режиму чернетки, та сервісний рівень обміну даними на базі Apollo Client. Такий підхід гарантує, що додавання будь-якого нового віджета (наприклад, календаря) вимагатиме лише створення нового React-компонента на фронтенді та додавання відповідного типу до GraphQL-схеми, абсолютно не зачіпаючи логіку авторизації чи роботи з базою даних.

2.2 Проєктування гібридної структури бази даних та конфігурації Fluent API

Перехід від концептуального проєктування модулів системи до їхньої безпосередньої технічної реалізації потребує детальної побудови схеми бази даних та опису логіки її взаємодії з додатком. Відповідно до обраної гібридної моделі збереження інформації (SQL + JSON), структура даних платформи у СУБД PostgreSQL розділена на два функціональні рівні: реляційне ядровий каркас та документо-орієнтоване сховище динамічних конфігурацій і табличних даних. Реляційна частина відповідає за надійну цілісність посилань, контроль прав доступу та деревоподібну структуру розділів користувача, тоді як JSON-поля бінарного типу JSONB акумулюють у собі внутрішній вміст віджетів та геометрію їхнього розташування на сітці інтерфейсу.

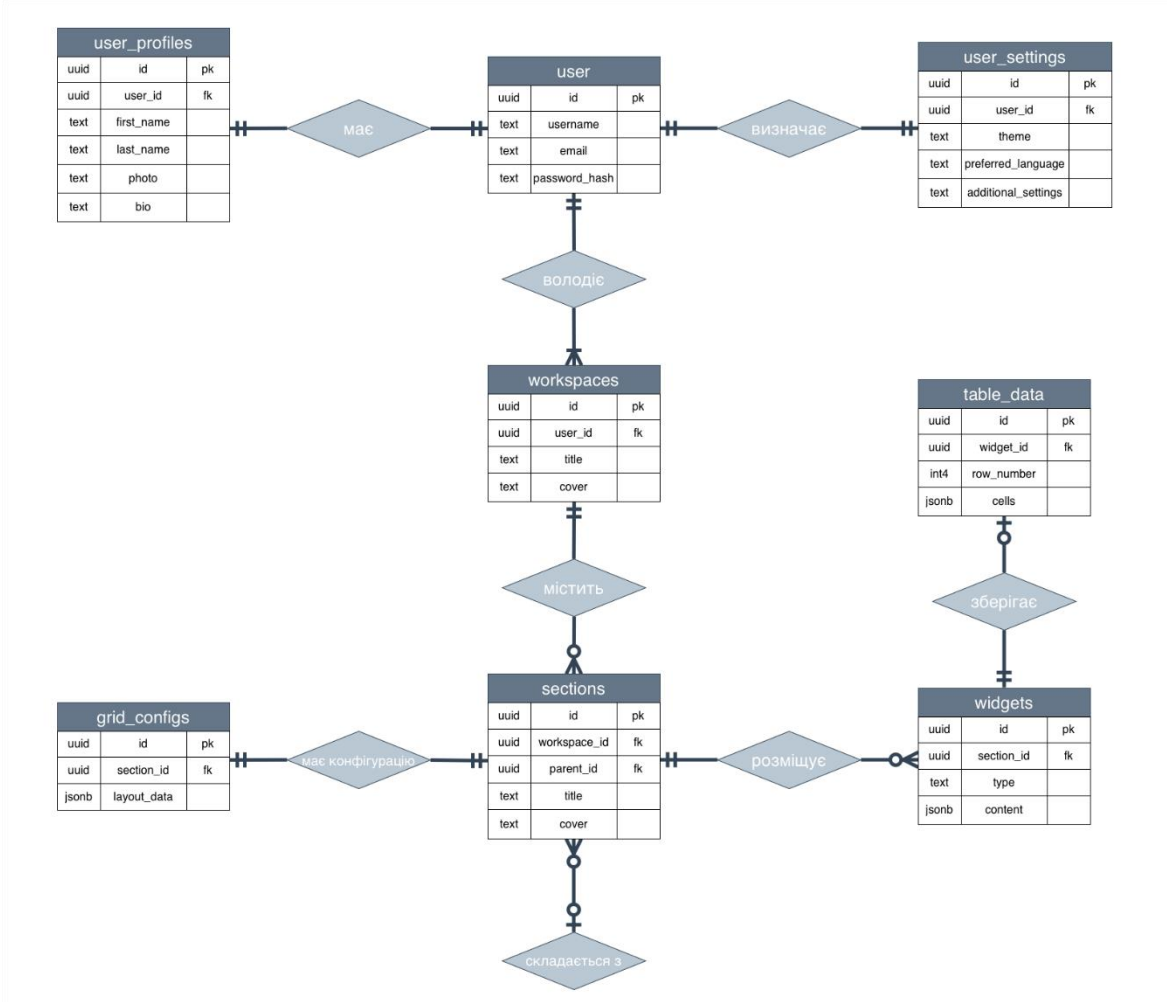

Рисунок 2.3 — ER-діаграма структури бази даних

Як показано на розробленій ER-діаграмі (рис. 2.3), центральне місце в архітектурі даних займає сутність користувача (таблиця users), яка виступає точкою входу для авторизації. Вона пов'язана відношенням «один до одного» із таблицями профілю (user_profiles) та персональними налаштуваннями (user_settings). Для організації простору користувача реалізовано відношення «один до багатьох» із таблицею workspaces. Кожен робочий простір містить у собі ієрархічне дерево розділів sections. Сутність розділів пов'язана відношенням «один до одного» з конфігурацією сітки (grid_configs) та «один до багатьох» із сутністю віджетів (widgets), яка, своєю чергою, зберігає табличні дані (table_data).

Для проєктування об'єктно-реляційного мапінгу (ORM) на базі фреймворку Entity Framework Core було розроблено базову абстракцію — клас BaseEntity. Він виносить на інфраструктурний рівень спільні для всіх таблиць атрибути: унікальний ідентифікатор запису (Guid Id), який на рівні PostgreSQL мапиться у тип uuid, а також поля аудиту — дату створення (CreateDate) та дату останнього редагування (UpdateDate). Налаштування цих полів інкапсульовано всередині статичного екшену OnModelCreating, що дозволяє автоматично застосовувати правила обов'язковості для всіх дочірніх таблиць системи за допомогою рефлексії, суттєво розвантажуючи конфігураційні класи та унеможливлюючи появу незаповнених аудит-полів.

Конфігурація унікальних обмежень та індексів для сутності користувача User реалізована через Fluent API з метою забезпечення високої безпеки та унікальності облікових даних. За допомогою перевизначеного статичного методу ініціалізації налаштовано композитні унікальні індекси (IsUnique) для полів Email та Username. Це гарантує захист бази даних від дублювання користувацьких записів ще на рівні СУБД, забезпечуючи швидкий пошук інформації під час проходження автентифікації.

```
/// <summary>
/// Abstract class for entities
/// </summary>
13 references | You, last month | 1 author (You)
public abstract class BaseEntity
{
    /// <summary>
    /// Entity ID
    /// </summary>
    [Key]
    [Column("id")]
    67 references
    public Guid Id { get; set; }

    /// <summary>
    /// Entity creation date (created_at)
    /// </summary>
    [Column("created_at")]
    1 reference
    public DateTime CreateDate { get; set; } = DateTime.MinValue;

    /// <summary>
    /// Entity update date (updated_at)
    /// </summary>
    [Column("updated_at")]
    1 reference
    public DateTime UpdateDate { get; set; } = DateTime.MinValue;

    [NotMapped]
    1 reference
    public static readonly Action<ModelBuilder> OnModelCreating = mb =>
    {
        foreach (var entityType in mb.Model.GetEntityTypes())
        {
            if (typeof(BaseEntity).IsAssignableFrom(entityType.ClrType))
            {
                mb.Entity(entityType.ClrType)
                    .Property("CreateDate").IsRequired();
                mb.Entity(entityType.ClrType)
                    .Property("UpdateDate").IsRequired();
            }
        }
    };
}
```

Рисунок 2.4 — Листинг реалізації базової сутності BaseEntity з Fluent API

Гнучкість та динамічність платформи забезпечується інтеграцією документо-орієнтованого типу JSONB у СУБД PostgreSQL, що наочно продемонстровано у структурі таблиці table_data (сутність TableData). Поле Cells, яке відповідає за динамічні клітинки користувацьких таблиць, спроєктовано з типом даних jsonb на рівні міграції. Це дозволяє повністю уникнути антипатерну EAV та жорстко заданих колонок: користувач може створювати таблиці будь-якої конфігурації, міняти типи колонок на фронтенді, а клієнтська частина надсилає та зчитує цілісні JSON-документи. СУБД PostgreSQL зберігає їх в оптимізованому бінарному форматі, що

забезпечує індексування внутрішніх полів JSON та надвисоку швидкість виконання операцій без переналаштування схеми таблиць.

```
10  [Table("users")]
    29 references
11  public class User : BaseEntity
12  {
13      /// <summary>
14      /// Unique username (users.username)
15      /// </summary>
16      [Column("username")]
        11 references
17      public string Username {get; set;} = string.Empty;
18
19      /// <summary>
20      /// User email (users.email)
21      /// </summary>
22      [Column("email")]
        11 references
23      public string Email {get; set;} = string.Empty;
24
25      /// <summary>
26      /// Password hash (users.password_hash)
27      /// </summary>
28      [Column("password_hash")]
        6 references
29      public string PasswordHash {get; set;} = string.Empty;
30
        1 reference
31      public virtual UserProfile Profile {get; set;}
32
        1 reference
33      public virtual UserSettings Settings {get; set;}
34
35      [NotMapped]
        0 references
36      public new static readonly Action<ModelBuilder> OnModelCreating = mb =>
37      {
38          mb.Entity<User>()
39              .HasIndex(x => x.Email)
40              .IsUnique();
41
42          mb.Entity<User>()
43              .HasIndex(x => x.Username)
44              .IsUnique();
45      };
```

Рисунок 2.5 — Листинг реалізації моделі User з Fluent API

Автоматизація рутинних процесів конфігурації СУБД є важливою ознакою масштабованої архітектури проєкту, що реалізовано всередині контексту AppDbContext через механізм рефлексії. Оскільки в системі використовуються перерахування (Enums), які мають кастомні атрибути мапінгу MapValueAttribute, у контексті розроблено метод RegisterEnumConverters. Він автоматично сканує всі сутності моделі, знаходить поля типу Enum, визначає за допомогою кастомного розширення ToMapValue цільовий тип збереження в базі даних та динамічно генерує об'єкти ValueConverter через вирази-дерева (Expression Trees), викликаючи generic-метод ConfigureEnumProperty. Такий підхід повністю позбавляє

необхідності вручну прописувати конвертацію для кожного нового перерахування при розробці, гарантуючи, що всі дані системних перерахувань (наприклад, типи віджетів WidgetType) зберігаються в базі у вигляді уніфікованих рядків або чисел автоматично.

Аналогічно на інфраструктурному рівні реалізовано метод ApplyGlobalConversions для полів типу DateTime та DateTime?. Цей метод за допомогою вбудованого ValueConverter автоматично приводить будь-яку дату до міжнародного стандарту UTC (DateTimeKind.Utc) перед її відправкою в базу даних PostgreSQL та коректно відновлює її специфікацію під час зчитування сервером. Це вирішує критичну проблему синхронізації часу в розподілених вебсистемах, коли клієнт і сервер можуть знаходитися в різних часових поясах.

2.3 Реалізація транзакційної логіки на базі патерну Unit of Work

Забезпечення атомарності, цілісності та консистентності даних під час виконання складних каскадних операцій (наприклад, створення робочого простору з автоматичною ініціалізацією головного розділу або каскадного видалення віджетів) є критично важливою вимогою до серверної архітектури. В нашої вебплатформі цю задачу вирішено за допомогою централізованого впровадження архітектурного патерну Unit of Work, який реалізовано у вигляді проміжного програмного забезпечення — custom Middleware. Таке рішення дозволяє повністю винести логіку фіксації транзакцій з окремих репозиторіїв та сервісів, автоматизуючи управління життєвим циклом даних на рівні кожного окремого HTTP-запиту.

Концепція роботи розробленого UnitOfWorkMiddleware базується на перехопленні контенту виконання запиту та аналізі результуючого статус-коду відповіді сервера. Під час надходження запиту Middleware отримує екземпляр контексту бази даних AppDbContext та інфраструктурний сервіс очищення ресурсів

IFileCleanupService через вбудований контейнер ін'єкції залежностей (Dependency Injection Scope).

Для візуалізації життєвого циклу запиту та черговості виклику сервісів під час фіксації транзакції було розроблено діаграму послідовності.

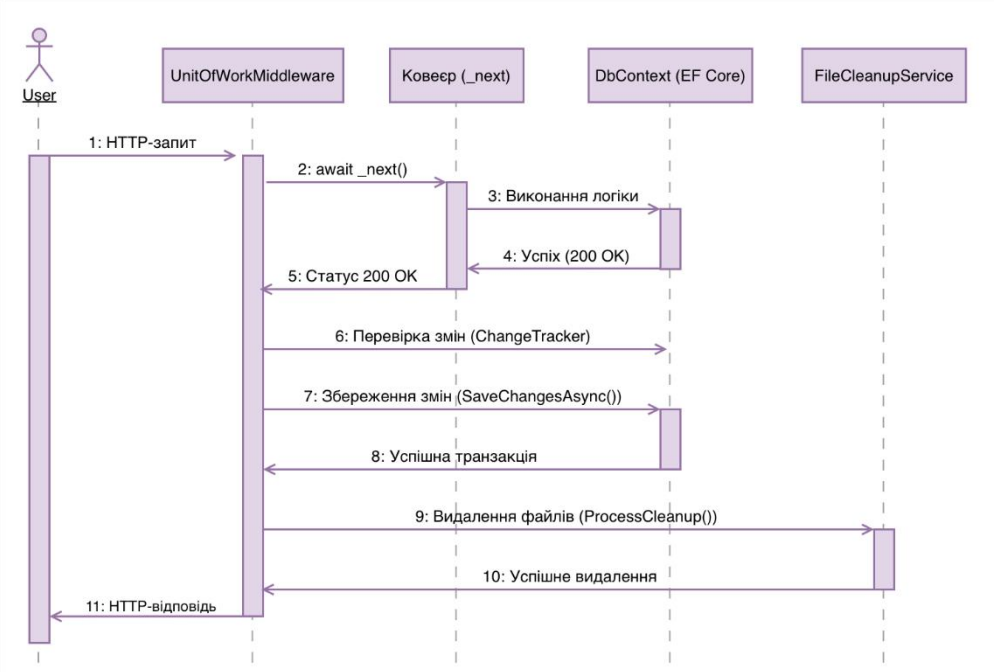

Рисунок 2.6 — Діаграма послідовності обробки запиту через UnitOfWorkMiddleware

Як показано на рис. 2.6, Middleware передає керування наступному компоненту конвеєра за допомогою виклику await _next(context). Якщо вся бізнес-логіка відпрацювала без помилок, і результуючий статус-код відповіді знаходиться в успішному діапазоні (від 200 до 299), Middleware ініціює процес синхронізації файлової системи, після чого викликає асинхронний метод SaveChangesAsync(), фіксуючи всі накопичені зміни в базі даних PostgreSQL. Якщо ж під час виконання запиту виникло виключення або статус-код вказує на помилку, збереження даних не відбувається, що гарантує захист системи від появи частково змінених чи некоректних даних .

Невіддільною частиною автоматизації транзакцій у нашій системі є аудит сутностей, реалізований через перевизначення методів SaveChanges та

SaveChangesAsync у класі AppDbContext. Перед безпосередньою трансляцією змін у SQL-команди, контекст ініціює приватний метод UpdateAuditEntities, який звертається до вбудованого механізму відстеження станів EF Core — ChangeTracker. Метод автоматично фільтрує всі сутності, успадковані від BaseEntity, що мають стан EntityState.Added або EntityState.Modified. Для нових записів поле CreateDate автоматично заповнюється поточним часом у форматі DateTime.UtcNow, а для редагованих — оновлюється поле UpdateDate, що повністю виключає людський фактор і гарантує точний аудит будь-якої дії в системі.

```
9   public class UnitOfWorkMiddleware
10  {
        2 references
11      private readonly RequestDelegate _next;
        0 references
12      public UnitOfWorkMiddleware(RequestDelegate next)
13      {
14          _next = next;
15      }
        0 references
16      public async Task InvokeAsync(HttpContext context)
17      {
18          var dbContext = context.RequestServices.GetRequiredService<AppDbContext>();
19
20          var cleanupService = context.RequestServices.GetRequiredService<IFileCleanupService>();
21
22          await _next(context);
23
24          if (context.Response.StatusCode >= 200 && context.Response.StatusCode < 300)
25          {
26              ExtractFilesToDelete(dbContext, cleanupService);
27
28              await dbContext.SaveChangesAsync();
29
30              cleanupService.ProcessCleanup();
31          }
32      }
        1 reference
33      private void ExtractFilesToDelete(AppDbContext dbContext, IFileCleanupService cleanupService)
34      {
35          var deletedWorkspaces = dbContext.ChangeTracker.Entries<Workspace>()
36              .Where(e => e.State == EntityState.Deleted)
37              .Select(e => e.Entity);
38
39          foreach (var workspace in deletedWorkspaces)
40          {
41              cleanupService.AddFileToDelete(workspace.Cover);
42          }
43
44          var deletedSections = dbContext.ChangeTracker.Entries<Section>()
45              .Where(e => e.State == EntityState.Deleted)
46              .Select(e => e.Entity);
47
48          foreach (var section in deletedSections)
49          {
50              cleanupService.AddFileToDelete(section.Cover);
51          }
52
53          var deletedWidgets = dbContext.ChangeTracker.Entries<Widget>()
54              .Where(e => e.State == EntityState.Deleted && e.Entity.Type == WidgetType.Image)
55              .Select(e => e.Entity);
56
57          foreach (var widget in deletedWidgets)
58          {
59              cleanupService.AddFileToDelete(widget.Content ?? string.Empty);
60          }
61      }
```

Рисунок 2.7 — Листинг реалізації UnitOfWorkMiddleware

Особливим архітектурним викликом при видаленні сутностей є синхронізація бази даних із фізичним сховищем файлів на сервері (папкою wwwroot/uploads), оскільки каскадне видалення в PostgreSQL очищає лише текстові посилання на обкладинки розділів чи картинки віджетів, залишаючи самі файли «привидами» на диску сервера. Для вирішення цієї проблеми в конвеєр транзакції інтегровано метод ExtractFilesToDelete, який перед фіксацією змін аналізує ChangeTracker. Якщо з бази видаляється об'єкт Workspace, Section або Widget із типом WidgetType.Image, Middleware вилучає відносні шляхи до їхніх файлів обкладинок чи контенту і реєструє їх у сервісі очищення за допомогою методу AddFileToDelete.

Фінальний етап очищення ресурсів виконується інфраструктурним сервісом FileCleanupService. Сервіс агрегує список файлів, відсікає дублікати за допомогою .Distinct() (що запобігає повторним спробам видалення одного й того самого файлу) і за допомогою безпечних конструкцій try-catch перевіряє наявність файлів на диску через File.Exists(path) та фізично видаляє їх за допомогою File.Delete(path).

```
3   public class FileCleanupService : IFileCleanupService
4   {
        3 references
5       private readonly ILogger<FileCleanupService> _logger;
        3 references
6       private readonly List<string> _filesToDelete = new();
7
        0 references
8       public FileCleanupService(ILogger<FileCleanupService> logger)
9       {
10          _logger = logger;
11      }
12
        4 references
13      public void AddFileToDelete(string relativePath)
14      {
15          if (!string.IsNullOrEmpty(relativePath))
16          {
17              var fullPath = System.IO.Path.Combine("wwwroot", "uploads", relativePath);
18              _filesToDelete.Add(fullPath);
19          }
20      }
21
        2 references
22      public void ProcessCleanup()
23      {
24          var distinctedFiles = _filesToDelete.Distinct().ToList();
25
26          foreach (var path in distinctedFiles)
27          {
28              try
29              {
30                  if (File.Exists(path))
31                  {
32                      File.Delete(path);
33                      _logger.LogInformation($"[CleanupService] The file has been successfully deleted: {path}");
34                  }
35              }
36              catch (Exception ex)
37              {
38                  _logger.LogWarning($"[CleanupService] Failed to delete the file: {path}. Error: {ex.Message}");
39              }
40          }
41
42          _filesToDelete.Clear();
43      }
```

Рисунок 2.8 — Листинг реалізації інфраструктурного сервісу FileCleanupService

Такий підхід гарантує абсолютну синхронізацію дискового простору та СУБД: фізичні файли видаляються з сервера строго після того, як транзакція успішно зафіксована в базі даних. Якщо видалення файлу зазнає невдачі через тимчасове блокування операційною системою або відсутність прав, сервіс не зупиняє роботу вебплатформи, а логує детальну помилку через ILogger.

2.4 Розробка GraphQL API для каскадного отримання та модифікації даних

Проєктні обмеження традиційного стилю REST API, які були проаналізовані у першому розділі роботи, зумовили необхідність розробки гнучкого, типізованого та оптимізованого шару клієнт-серверної взаємодії. Для нашої вебплатформи цю задачу вирішено шляхом розробки GraphQL API на базі сучасного high-performance сервера Hot Chocolate. Головною інженерною перевагою побудованого API є надання клієнтській частині (React) можливості самостійно конструювати граф запиту для отримання ієрархічних структур даних (наприклад, простору разом із вкладеними розділами та конфігураціями їхніх віджетів) в межах одного єдиного мережевого звернення.

Архітектура GraphQL-сервера платформи побудована за принципом декомпозиції корінних точок входу (Query та Mutation). Замість створення монолітних класів запитів та модифікацій, які містили б у собі десятки методів для всіх сутностей системи, було реалізовано патерн розділення за доменами бізнес-логіки. Кореневий клас Query виступає декларативним диспетчером, що перенаправляє запити на ізольовані класи-сервіси сутностей. Для забезпечення залізобетонного рівня безпеки, майже всі вузли графа (окрім публічних словників та мутацій авторизації) захищені декларативним атрибутом безпеки [Authorize], інтегрованим з інфраструктурою JWT-автентифікації ASP.NET Core.

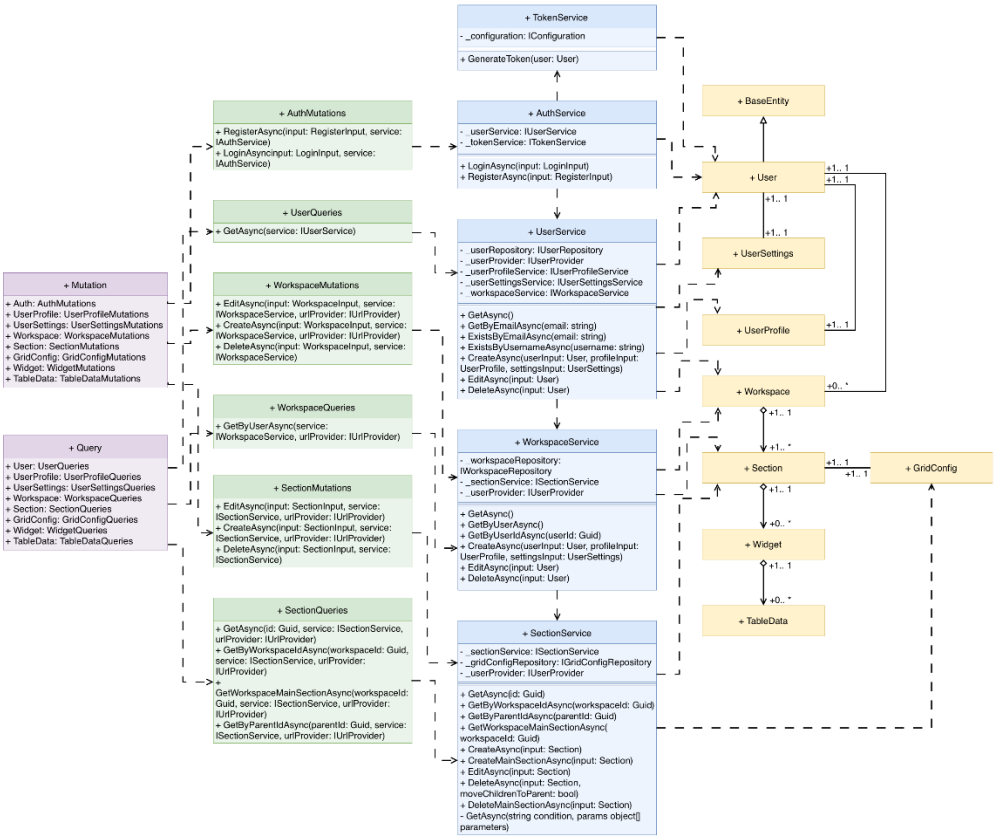

Рисунок 2.9 — Діаграма класів серверної частини

На розробленій діаграмі класів (рис. 2.9) представлено модель серверної частини вебплатформи, де всі компоненти чітко розмежовані за трьома логічними рівнями: доменні сутності, прикладні бізнес-сервіси та шар обробки GraphQL-запитів. Рівень даних базується на абстрактному класі BaseEntity, від якого успадковуються всі моделі. Сутність користувача User пов'язана відношеннями «один до одного» з профілем (UserProfile) та налаштуваннями (UserSettings), а також має відношення кратності «один до нуля або більше» (1:0..*) із робочими просторами (Workspace). Структурні зв'язки між простором, сторінками та контентом організовано за допомогою відношень агрегації: клас Workspace містить ієрархічні розділи (Section), які виступають агрегаторами для віджетів (Widget), які, своєю чергою, каскадно володіють даними таблиць (TableData).Також сутність «Section» має відношення «один до одного» з конфігурацією сітки (GridConfig).

Шар прикладних сервісів інкапсулює бізнес-логіку платформи та взаємодіє з доменними моделями через наскрізні зв'язки залежності. Зокрема, TokenService здійснює генерацію криптографічних JWT-сесій на основі сутності User, а AuthService координує процеси входу, використовуючи можливості TokenService та UserService. Логіка управління профілем користувача, робочими просторами та ієрархією розділів покладена на UserService, WorkspaceService та SectionService відповідно, які послідовно взаємодіють між собою та операційно залежать від відповідних сутностей бази даних. Зовнішній інтерфейс взаємодії з клієнтом реалізовано на рівні GraphQL-типів Hot Chocolate, де класи модифікації даних (AuthMutations, WorkspaceMutations, SectionMutations) та класи читання (UserQueries, WorkspaceQueries, SectionQueries) через слабкопов'язані відношення залежності викликають методи прикладних сервісів, після чого централізовано агрегуються у загальні кореневі вузли Mutation та Query відповідно

```
5   public class Query
6   {
7       [Authorize]
        0 references
8       public UserQueries User => new();
9
10      [Authorize]
        0 references
11      public UserProfileQueries UserProfile => new();
12
13      [Authorize]
        0 references
14      public UserSettingsQueries UserSettings => new();
15
16      [Authorize]
        0 references
17      public WorkspaceQueries Workspace => new();
18
19      [Authorize]
        0 references
20      public SectionQueries Section => new();
21
22      [Authorize]
        0 references
23      public GridConfigQueries GridConfig => new();
24
25      [Authorize]
        0 references
26      public WidgetQueries Widget => new();
27
28      [Authorize]
        0 references
29      public TableDataQueries TableData => new();
30
        0 references
31      public DictionaryQueries Dictionaries => new();
32
33  }
```

Рисунок 2.10 — Листинг структури кореневого класу Query

```
6   public class Mutation
7   {
        0 references
8       public AuthMutations Auth => new();
9
10      [Authorize]
        0 references
11      public UserProfileMutations UserProfile => new();
12
13      [Authorize]
        0 references
14      public UserSettingsMutations UserSettings => new();
15
16      [Authorize]
        0 references
17      public WorkspaceMutations Workspace => new();
18
19      [Authorize]
        0 references
20      public SectionMutations Section => new();
21
22      [Authorize]
        0 references
23      public GridConfigMutations GridConfig => new();
24
25      [Authorize]
        0 references
26      public WidgetMutations Widget => new();
27
28      [Authorize]
        0 references
29      public TableDataMutations TableData => new();
30  }
```

Рисунок 2.11 — Листинг структури кореневого класу Mutation

Практична реалізація бізнес-мутацій та запитів продемонстрована на прикладі управління ієрархічними розділами (класи SectionQueries та SectionMutations). Кожен метод GraphQL-шару є тонким провідником: він приймає вхідні аргументи, за допомогою механізму Dependency Injection через атрибут [Service] отримує потрібний екземпляр сервісу бізнес-логіки, делегує йому виконання операції та транслює доменну сутність (Entity) у безпечний об'єкт передачі даних (DTO) за допомогою методу ToDto(urlProvider).

```
8   public class SectionMutations
9   {
10      [Authorize]
11      [GraphQLName("edit")]
            0 references
12      public async Task<SectionDto?> EditAsync(
13          SectionInput input,
14          [Service] ISectionService service,
15          [Service] IUrlProvider urlProvider)
16      {
17          var section = await service.EditAsync(input.ToEntity());
18
19          return section?.ToDto(urlProvider);
20      }
21
22      [Authorize]
23      [GraphQLName("create")]
            0 references
24      public async Task<SectionDto?> CreateAsync(
25          SectionInput input,
26          [Service] ISectionService service,
27          [Service] IUrlProvider urlProvider)
28      {
29          var section = await service.CreateAsync(input.ToEntity());
30
31          return section?.ToDto(urlProvider);
32      }
33
34      [Authorize]
35      [GraphQLName("delete")]
            0 references
36      public async Task<bool> DeleteAsync(
37          SectionInput input,
38          bool moveChildrenToParent,
39          [Service] ISectionService service)
40      {
41          await service.DeleteAsync(input.ToEntity(), moveChildrenToParent);
42
43          return true;
44      }
45  }
```

Рисунок 2.12 — Листинг реалізації бізнес-мутацій для сутності Section

Важливою архітектурною перевагою побудованого API є надійна валідація вхідних даних на етапі парсингу GraphQL-запиту. Завдяки інтеграції пакета AppAny.HotChocolate.FluentValidation, конвеєр GraphQL сервера перехоплює вхідні DTO (наприклад, LoginInput або RegisterInput), автоматично знаходить відповідний інжектований валідатор, успадкований від AbstractValidator<T>, та виконує перевірку бізнес-правил. У разі невідповідності даних правилам, Hot Chocolate миттєво перериває виконання запиту та формує типізований масив помилок errors із унікальними кодами (наприклад

ValidationErrors.InvalidEmailCode), що дозволяє фронтенду локалізовано виводити помилки користувачам в інтерфейсі.

```
8   public class SectionQueries
9   {
10      [Authorize]
11      [GraphQLName("get")]
            0 references
12      public async Task<SectionDto?> GetAsync(
13          Guid id,
14          [Service] ISectionService service,
15          [Service] IUrlProvider urlProvider)
16      {
17          var section = await service.GetAsync(id);
18          return section?.ToDto(urlProvider);
19      }
20
21      [Authorize]
22      [GraphQLName("getByWorkspaceId")]
            0 references
23      public async Task<List<SectionDto>> GetByWorkspaceIdAsync(
24          Guid workspaceId,
25          [Service] ISectionService service,
26          [Service] IUrlProvider urlProvider)
27      {
28          var sections = await service.GetByWorkspaceIdAsync(workspaceId);
29          return sections.Select(x => x.ToDto(urlProvider)).ToList();
30      }
31
32      [Authorize]
33      [GraphQLName("getWorkspaceMainSection")]
            0 references
34      public async Task<SectionDto?> GetWorkspaceMainSectionAsync(
35          Guid workspaceId,
36          [Service] ISectionService service,
37          [Service] IUrlProvider urlProvider)
38      {
39          var section = await service.GetWorkspaceMainSectionAsync(workspaceId);
40          return section?.ToDto(urlProvider);
41      }
42
43      [Authorize]
44      [GraphQLName("getByParentId")]
            0 references
45      public async Task<List<SectionDto>> GetByParentIdAsync(
46          Guid parentId,
47          [Service] ISectionService service,
48          [Service] IUrlProvider urlProvider)
49      {
50          var sections = await service.GetByParentIdAsync(parentId);
51          return sections.Select(x => x.ToDto(urlProvider)).ToList();
52      }
53  }
```

Рисунок 2.13 — Листинг реалізації запитів для сутності Section

Для забезпечення безпеки та приховування внутрішніх деталей реалізації бекенду (таких як трасування стек-викликів чи системні повідомлення бази даних) у середовищі production, було розроблено та впроваджено глобальний фільтр обробки помилок — GraphQLErrorFilter, що реалізує інтерфейс IErrorFilter. Його

завдання — аналізувати всі виключення, що виникають під час виконання GraphQL-операцій.

Логіка роботи GraphQLErrorFilter побудована таким чином: якщо виключення належить до типу кастомних доменних помилок системи (BaseException), фільтр перехоплює його, вилучає зрозуміле для користувача повідомлення та присвоює системний бізнес-код помилки (ErrorCode). Якщо помилка є результатом порушення правил FluentValidation, вона пропускається без змін для збереження контексту валідації. Для всіх інших непередбачуваних критичних збоїв фільтр перевіряє директиви препроцесора: у режимі розробки (#if DEBUG) деталі помилки залишаються відкритими для полегшення дебагу, а у релізі — повністю маскуються уніфікованим повідомленням та кодом INTERNAL_SERVER_ERROR.

```
5   public class GraphQLErrorFilter : IErrorFilter
6   {
        0 references
7       public IError OnError(IError error)
8       {
9           if (error.Exception is BaseException ex)
10          {
11              return error.WithMessage(ex.Message)
12                  .WithCode(ex.ErrorCode);
13          }
14
15          if (ValidationErrors.GetAllErrorCodes().Contains(error.Code ?? string.Empty))
16          {
17              return error;
18          }
19
20          #if DEBUG
21              return error;
22          #else
23              return error.WithMessage("Сталася внутрішня помилка сервера.")
24                  .WithCode("INTERNAL_SERVER_ERROR");
25          #endif
26      }
27  }
```

Рисунок 2.14 — Листинг реалізації глобального фільтра помилок GraphQLErrorFilter

Фінальне конфігурування та збирання GraphQL-сервера виконується у файлі Program.cs за допомогою розширення AddGraphQLServer(). У конвеєр обробки інжектуються типи запитів, мутацій, фільтр помилок, а також глобальні сервіси фільтрації (AddFiltering()) та сортування (AddSorting()). Конфігурація області

ін'єкції залежностей DependencyInjectionScope.Request гарантує, що для кожної мутації та запиту створюється ізольований scoped-контекст, що забезпечує стабільну паралельну роботу додатку та повну сумісність із транзакційним шаром UnitOfWorkMiddleware.

2.5 Реалізація механізму автентифікації користувачів

Забезпечення безпеки персонального простору, конфіденційності користувацьких даних та ізоляції воркспейсів потребує проектування надійної підсистеми автентифікації та авторизації. В нашій вебплатформі безпеку клієнт-серверної взаємодії реалізовано на основі міжнародного стандарту JWT (JSON Web Tokens) у синергії з криптостійким хешуванням паролів за допомогою алгоритму BCrypt. Така архітектура реалізує концепцію Stateless авторизації, що дозволяє серверу не зберігати сесії в оперативній пам'яті, а повністю валідувати користувацькі права на основі криптографічного підпису токена.

Процес реєстрації та входу користувачів до системи ініціюється на рівні GraphQL-шару через мутаційний тип AuthMutations. Завдяки декларативним атрибутам Hot Chocolate, точки входу login та register інтегровані з конвеєром [UseFluentValidation]. Це гарантує, що сирі дані не потраплять до сервісів бізнес-логіки, якщо вони не пройдуть сувору перевірку валідаторів LoginInputValidator та RegisterInputValidator. На етапі реєстрації система в обов'язково перевіряє унікальність електронної пошти та користувацького імені в базі даних за допомогою методів ExistsByEmailAsync та ExistsByUsernameAsync класу UserService.

У разі успішної перевірки унікальності, підсистема безпеки забезпечує криптографічний захист пароля. Зберігання паролів у відкритому вигляді є критичним порушенням безпеки, тому в сервісі AuthService застосовано метод BCrypt.Net.BCrypt.HashPassword. Алгоритм BCrypt використовує адаптивну функцію хешування на основі шифру Blowfish, яка автоматично додає випадкову

криптографічну сіль (Salt) та виконує декілька етапів шифрування. Це повністю нівелює можливість зламу пароля за допомогою заздалегідь обчислених Rainbow Tables або методів грубої сили (Brute-force). Під час входу користувачів (LoginAsync) сервіс порівнює введений пароль із хешем у базі даних за допомогою методу BCrypt.Verify.

Листинг реалізації бізнес-логіки автентифікації, реєстрації та хешування у сервісі AuthService подано в додатку А.

Генерація безпечного криптографічного токена покладена на сервіс TokenService. Метод GenerateToken формує набір тверджень (Claims), які інкапсулюють у собі публічну інформацію про користувачів: унікальний ідентифікатор (JwtRegisteredClaimNames.Sub), пошту та його користувацьке ім’я. Підпис токена виконується на основі секретного ключа Jwt:Key, зчитаного з конфігураційного файлу сервера, із застосуванням симетричного алгоритму шифрування HmacSha256. Життєвий цикл токена обмежено 7 добами (DateTime.UtcNow.AddDays(7)), що мінімізує ризики у разі його перехоплення зловмисником.

Особливим досягненням розробленої підсистеми є вирішення проблеми відсутності контексту користувача під час транзакційного створення акаунта. Процес створення користувача у методі CreateAsync є комплексним: спочатку запис фіксується в таблиці users, після чого каскадно створюється профіль UserProfile, системні налаштування UserSettings та первинний робочий простір користувача Workspace з назвою «My workspace» (який, своєю чергою, автоматично генерує головний розділ Section). Оскільки всі ці супутні інфраструктурні сервіси під час виклику методу CreateAsync звертаються до провайдера доступу для отримання ID поточного користувача, а в HTTP-контексті токена ще фізично немає, було спроєктовано інтерфейс IUserProvider та його реалізацію для вебплатформи WebUserProvider.

```
3   namespace Lifield.Api.Common.Providers;
4
    2 references | You, 4 weeks ago | 1 author (You)
5   public class WebUserProvider : IUserProvider
6   {
        2 references
7       private readonly IHttpContextAccessor _accessor;
        3 references
8       private Guid? _registrationUserId;
9
        0 references
10      public WebUserProvider(IHttpContextAccessor accessor)
11      {
12          _accessor = accessor;
13      }
14
        2 references
15      public void SetRegistrationContext(Guid userId)
16      {
17          if (userId != Guid.Empty)
18          {
19              _registrationUserId = userId;
20          }
21      }
22
        16 references
23      public Guid GetCurrentUserId(bool allowRegistrationId = false)
24      {
25          var rawUserId = _accessor.HttpContext?.User?.FindFirst(ClaimTypes.NameIdentifier)?.Value;
26
27          if (rawUserId == null || !Guid.TryParse(rawUserId, out var userId))
28          {
29              return allowRegistrationId && _registrationUserId.HasValue
30                  ? _registrationUserId.Value
31                  : Guid.Empty;
32          }
33
34          return userId;
35      }
36  }
```

Рисунок 2.15 — Листинг реалізації Claims-розширень та контекстного провайдера WebUserProvider

Для вилучення ідентифікатора у звичайних GraphQL-запитах, які захищені політиками авторизації, розроблено статичний метод розширення GetUserId для класу ClaimsPrincipal. Метод безпечно витягує клейм ClaimTypes.NameIdentifier, виконує валідацію структури Guid.TryParse та, у разі підміни токена або порожнього значення, генерує виключення InvalidUserIdException, яке миттєво перехоплюється фільтром помилок GraphQLErrorFilter та маскується на інфраструктурному рівні. Конфігурація авторизаційного шару закривається у файлі Program.cs за допомогою методів AddAuthentication та AddJwtBearer, що зв'язує серверні валідаційні параметри TokenValidationParameters з GraphQL-сервером через інжектовану команду AddAuthorization().

2.6 Розробка клієнтської частини конструктора інтерфейсів на React

Практична реалізація клієнтської частини нашої вебплатформи підпорядкована концепції односторінкових застосунків (SPA), де все обчислювальне навантаження з рендерингу графічних елементів, валідації користувацького введення та управління станом інтерфейсу перенесене безпосередньо у браузер користувача. Такий архітектурний підхід дозволяє уникнути класичної проблеми монолітних систем — постійного перезавантаження сторінок та надлишкового рендерингу HTML-розмітки на стороні сервера. Клієнтська частина платформи функціонує як незалежний застосунок, розгорнутий на базі фреймворку React, який взаємодіє з сервером .NET виключно через асинхронні структуровані запити, забезпечуючи плавну роботу інтерфейсу на рівні 60 FPS.

Центральним інфраструктурним ядром, яке забезпечує взаємодію між клієнтом та GraphQL-сервером, виступає інстанційований та сконфігурований клієнт Apollo Client. Для інтеграції цього інструменту в життєвий цикл React-застосунку весь корінь декларативного дерева компонентів у файлі main.jsx було обгорнуто в провайдер контексту <ApolloProvider client="{client}">. Це архітектурне рішення надає будь-якому дочірньому компоненту системи (наприклад, сторінці конструктора розділу SectionPage) безперешкодний доступ до методів виконання запитів useQuery та мутацій useMutation, повністю абстрагуючи від ручного керування HTTP-сесіями, прошивки мережевих заголовків та низькорівневого парсингу отриманих JSON-пакетів.

Особливо важливо звернути увагу на реалізацію концепції ланцюжка посилань (Link Chain) розробленого модуля клієнта apolloClient.js, яка дозволяє перехоплювати, модифікувати та розширювати GraphQL-запити безпосередньо перед їх відправкою в мережу. Базова конфігурація зв'язку складається з поєднання двох функціональних рівнів: транспортного лінка HttpLink, який вказує на локальну точку доступу сервера http://localhost:5140/graphql/, та контекстного лінка

безпеки authLink, створеного за допомогою фабричного методу setContext. Ланцюжок формується шляхом композиції .concat(), де запит спочатку проходить крізь фільтр авторизації, і лише потім транслюється в бінарний HTTP-потік мережевого адаптера.

Логіка роботи перехоплювача authLink забезпечує залізобетонну безпеку системи та реалізує механізм автентифікації користувача на кожному кроці взаємодії. Перед відправкою будь-якої операції, перехоплювач асинхронно звертається до локального сховища браузера користувача для перевірки наявності захищеної сесії за допомогою команди localStorage.getItem('authToken'). Якщо токен присутній, лінк динамічно модифікує стандартний HTTP-заголовок, додаючи в нього параметр Authorization: Bearer <token>, що дозволяє серверному Middleware авторизації успішно валідувати цифровий підпис та ідентифікувати користувача. Якщо сховище порожнє, заголовок залишається порожнім, забезпечуючи безперешкодний доступ лише до публічних методів реєстрації та входу.

Для оптимізації роботи з даними та запобігання надлишковим повторним мережевим запитам, Apollo Client використовує вбудовану систему кешування InMemoryCache. Ця технологія автоматично нормалізує отримані від сервера GraphQL-відповіді, розбиваючи їх на окремі об'єкти за унікальними парами полів, і зберігає їх у внутрішній структурі оперативної пам'яті клієнта. У поєднанні з гнучкими стратегіями зчитування даних, такими як fetch-policy: 'cache-and-network', застосунок демонструє колосальну швидкість роботи: система миттєво рендерить сторінку користувачу, використовуючи локальний кеш, і одночасно у фоновому режимі робить запит до СУБД PostgreSQL для актуалізації контенту, мінімізуючи візуальну затримку інтерфейсу.

Окрім безпеки, інфраструктурний міст Apollo Client бере на себе задачу глобальної інтернаціоналізації (i18n) клієнт-серверного обміну даними. Оскільки платформа підтримує динамічне перемикання мовних культур (українська та англійська), бекенд має повертати певні дані відповідною мовою. Локалізація

статики на фронтенді базується на екосистемі i18next, конфігурація якої (файл i18n/config.js) використовує плагін LanguageDetector для автоматичного визначення пріоритетної культури користувачів на основі аналізу localStorage та налаштувань ОС. Для зв'язку з бекендом у authLink інтегровано зчитування активної мови інтерфейсу за допомогою властивості i18n.language. Лінк прошиває її в HTTP-заголовок "Accept-Language", завдяки чому на клієнта повертаються дані мовою користувачів в межах однієї сесії без необхідності перезавантаження сторінки.

```
6   const httpLink = new HttpLink({
7       uri: 'http://localhost:5140/graphql/',
8   });
9
10  const authLink = setContext((_, { headers}) => {
11      const token = localStorage.getItem('authToken');
12      const language = i18n.language || 'en';
13
14      return {
15          headers: {
16              ...headers,
17              authorization: token ? `Bearer ${token}` : "",
18              "Accept-Language": language
19          }
20      }
21  });
22
23  export const client = new ApolloClient({
24      link: authLink.concat(httpLink),
25      cache: new InMemoryCache(),
26  });
```

Рисунок 2.16 — Листинг конфігурації інфраструктурного шару Apollo Client з конвеєром AuthLink та HttpLink

Синхронізація глобальних користувацьких параметрів (таких як колірна тема та мова, які ми закладали в структуру бази даних) на рівні ініціалізації клієнта реалізована за допомогою кастомних React-хуків useUserSettings та useLanguageSync. Хук useUserSettings ініціює GraphQL-запит GET_USER_SETTINGS зі стратегією network-only строго після успішної

авторизації (контролюється за допомогою інструкції умовного пропуску запиту skip: !localStorage.getItem('authToken')).

Отримані дані передаються в хук синхронізації, який взаємодіє з бібліотекою локалізації. Для запобігання десинхронізації інтерфейсу, корінь застосунку App.jsx реалізує динамічне перевизначення унікального ключа контенту за допомогою властивості . Це змушує React виконати чистий та безпечний перерендер лише мовно-залежних вузлів дерева компонентів під час зміни налаштувань, що забезпечує стабільність клієнтського UI та повну консистентність стану застосунку.

2.6.1 Реалізація динамічної 24-колонкової сітки та режиму редагування

Композиційною та візуальною основою конструктора інтерфейсів в нашій вебплатформі є модульна сітка, яка реалізована на клієнтському рівні за допомогою спеціальної бібліотеки react-grid-layout. Для забезпечення гарної точності, гнучкості та кастомізації графічного простору було спроєктовано 24-колонкову динамічну модель сітки. Збільшення кількості колонок до 24 є важливим рішенням, яке вирішує проблему грубого позиціонування елементів: це дозволяє користувачам створювати зовсім дрібні віджети (наприклад, одиночні віджети іконок розміром в 1 клітинку сітки), розташовувати кілька різнопланових блоків на одному горизонтальному рівні та масштабувати таблиці без порушення композиції візуального простору.

Для того щоб сітка залишалася повністю адаптивною та коректно реагувала на фізичні зміни розмірів користувацького екрана, в компоненті SectionDisplay.jsx реалізовано механізм автоматичного відстеження геометрії контейнера за допомогою референсу containerRef та вбудованого спостерігача ResizeObserver. Система динамічно розраховує поточну доступну ширину в пікселях (offsetWidth) та пропорційно обчислює ширину однієї умовної колонки.

При цьому висота рядків сітки (dynamicRowHeight) дорівнює вирахованій ширині колонки, а зовнішні та внутрішні відступи між елементами

(dynamicMargin) адаптуються залежно від розмірів екрану. Таке математичне співвідношення гарантує гнучкість інтерфейсу та збереження пропорцій віджетів на будь-яких пристроях.

Управління станом конструктора сторінок базується на строгому розмежуванні життєвого циклу компонента на два взаємовиключні стани: Режим перегляду та Режим редагування. Контроль цих станів реалізовано через локальний React-стейт isEditMode. Коли сторінка перебуває в режимі перегляду, метод applyModeSettings автоматично маркує всі елементи в масиві макету як static: true, а прапорці перетягування isDraggable та зміни розмірів isResizable примусово відключаються. Це повністю знімає навантаження з графічного движка браузера під час звичайного читання даних, оскільки система не відстежує координати курсора миші.

При активації режиму редагування компонент кардинально змінює свою поведінку та підключає додатковий інструментарій:

– візуально активується фонова сітка через динамічний SVG-паттерн заднього фону, параметри якого перераховуються на основі columnWidth та dynamicMargin;

– для кожного віджета властивість static змінюється на false, активуючи технологію Drag-and-Drop;

– з'являється бічна панель інструментів SectionSidebar для додавання нових віджетів;

Для запобігання критичного перевантаження мережі та бази даних, у конструкторі реалізовано механізм локальних чернеток, який керується через комплексний об'єкт стану pendingChanges. Будь-які маніпуляції користувачів в режимі редагування (перетягування блоків, додавання нових віджетів через handleAddWidget або їхнє локальне видалення) не відправляються на сервер миттєво, а записуються у тимчасових локальних масивах (newSections, updatedWidgets, newTableData тощо).

Синхронізація з базою даних PostgreSQL відбувається каскадно й атомарно лише під час виклику методу handleFinalSubmit. Всередині цього методу запускається складний транзакційний ланцюжок: спочатку асинхронно завантажуються фізичні медіафайли через REST API, потім виконуються послідовні GraphQL-мутації створення та оновлення розділів і віджетів, формується мапа відповідності ідентифікаторів (idMap), після чого одним масивом фіксуються рядки таблиць TableData та оновлений JSON-рядок конфігурації сітки saveGrid.

Листинг реалізації розрахунку параметрів 24-колонкової сітки та обробки подій у SectionDisplay подано в додатку Б.

Важливим елементом надійності роботи 24-колонкової сітки є алгоритм ущільнення, вбудований у ядро react-grid-layout. Координатна сітка оперує параметрами x (початкова колонка від 0 до 23), y (номер рядка), w (ширина в колонках) та h (висота в рядках). Під час перетягування блоків система автоматично зміщує нижні елементи по вертикальній осі Y у разі виникнення колізій, проте суворо блокує вихід будь-якого віджета за межі максимальної ширини у 24 колонки. Це гарантує залізобетонну монолітність макету, а будь-які геометричні зсуви розв'язуються клієнтським рушієм миттєво та плавно.

2.6.2 Побудова фабрики компонентів та алгоритму валідації

Поліморфний рендеринг графічних модулів усередині 24-колонкової сітки конструктора забезпечується впровадженням на клієнтському рівні архітектурного патерну «Фабрика компонентів» (Component Factory). Оскільки кожна секція простору може містити необмежену кількість різнотипних елементів, каркас сторінки не повинен мати жорстких залежностей від конкретних реалізацій віджетів. Зберігання структури в базі даних у вигляді узагальненого переліку сутностей із поліморфним полем Type вимагає від фронтенд-частини наявності гнучкого інтерпретатора, який здатний динамічно створити потрібний компонент у реальному часі.

Центральним елементом цієї архітектури виступає фабричний метод renderWidgetContent, інтегрований у загальний конвеєр рендерингу сторінки. На основі аналізу конфігураційного об'єкта віджета, отриманого через GraphQL API, фабрика виконує умовне розгалуження за допомогою конструкції switch(item.type).

Залежно від зчитаного типу, фабрика динамічно монтує у відповідний вузол сітки спеціалізований React-компонент: DividerWidget (розділювач), TextWidget (текст), ImageWidget (зображення), IconWidget (іконка), LinkWidget (посилання), TableWidget (таблиця), KanbanWidget (канбан-дошка) або ChartWidget (графік). Такий підхід повністю ізолює бізнес-логіку та внутрішній стейт конкретного віджета від глобального каркаса сітки, забезпечуючи високу масштабованість системи: додавання нового типу віджета вимагає лише створення його ізольованого компонента та реєстрації нової інструкції у фабричному методі.

Управління персональними налаштуваннями, контентом та метаданими віджетів зроблено через універсальний діалоговий інтерфейс WidgetModal.jsx. Цей компонент виконує роль фабрики керуючих форм. Аналізуючи тип обраного для редагування віджета, модальне вікно динамічно підбирає оптимальний розмір вікна (наприклад, max-w-md для тексту чи іконок та max-w-4xl для роботи зі таблицями) і рендерить відповідну форму налаштувань (TableForm, KanbanForm, TextForm тощо).

Глибока інтеграція структурованих даних на клієнтському рівні наочно продемонстрована на прикладі форми TableForm та віджета TableWidget. Для реалізації складних реляційних таблиць віджет інкапсулює у собі дворівневу структуру: метадані конфігурації (назва таблиці та назви колонок) зберігаються у вигляді серіалізованого JSON-рядка в полі Content моделі віджета, а самі клітинки даних мапляться на незалежні реляційні рядки TableData з прив'язкою до ідентифікатора віджета й сортуються за індексом rowNumber.

Листинг реалізації форми TableForm з організацією стовпчиків та рядків даних подано в додатку В.

Важливою частиною стабільності клієнтського інтерфейсу є алгоритм дворівневої ізольованої валідації. Він повністю розподіляє зони відповідальності між розмірами віджетів та введеними даними користувачами:

1. Перший рівень (Валідація розмірів): Реалізований на рівні глобального конфігуратора сітки WIDGETS_GRID_CONFIG. Для кожного типу віджета жорстко зафіксовані ліміти граничних розмірів (minW, minH, maxW, maxH). Наприклад, для текстового блоку TextWidget встановлено мінімальні розміри minW: 2, minH: 1, тоді як для графіка ChartWidget або таблиці TableWidget ліміти становлять minW: 6, minH: 3. Це унеможливлює ситуацію, коли користувач під час інтерактивного редагування простору може випадково стиснути таблицю чи дошку до нечитабельного стану розміром в одну клітинку, що порушило б цілісність інтерфейсу.

2. Другий рівень (Валідація введених даних): Реалізований у внутрішній логіці кожної фабричної форми всередині WidgetModal. На прикладі компонента TableForm, під час перемикання на вкладку з даними (activeTab === "data"), система через метод handleCellChange здійснює постійну локальну перевірку цілісності індексів та динамічне перерахування внутрішніх масивів. Форма самостійно контролює структуру клітинок, фільтрує порожні об'єкти та гарантує, що при виклику handleFormSubmit у локальний буфер чернеток (pendingChanges) потрапить очищений, структурований масив об'єктів, який повністю відповідає очікуванням схеми GraphQL.

Завдяки переносу валідації на сторону клієнта, серверний застосунок повністю звільняється від необхідності розбору синтаксису сирих клітинкових масивів, що зводить до мінімуму додаткові витрати на передачу даних та збільшує загальну продуктивність і швидкість відклику вебплатформи.

2.7 Опис компонентів інтерфейсу та демонстрація роботи системи

Результатом успішного проєктування, реалізації об'єктно-реляційного мапінгу та розробки клієнтської архітектури на базі React є готовий до використання прототип вебплатформи. Для підтвердження працездатності системи, валідації впроваджених рішень та оцінки зручності взаємодії було проведено розгортання та демонстраційне тестування розроблених модулів у браузері. На цьому етапі було проаналізовано ключові користувацькі сценарії, швидкість рендерингу елементів 24-колонкової сітки та коректність обробки поліморфних даних.

Процес взаємодії користувача із платформою розпочинається з екрана автентифікації. Форма входу та реєстрації розроблена з урахуванням сучасних стандартів ергономіки. Завдяки впровадженню клієнтських валідаторів, інтерфейс миттєво реагує на дії користувачів, блокуючи відправку GraphQL-мутацій у разі некоректного формату електронної пошти чи недостатньої довжини пароля, що суттєво знижує навантаження на мережевий канал системи. Візуальне представлення форми входу продемонстровано на рис. 2.17.

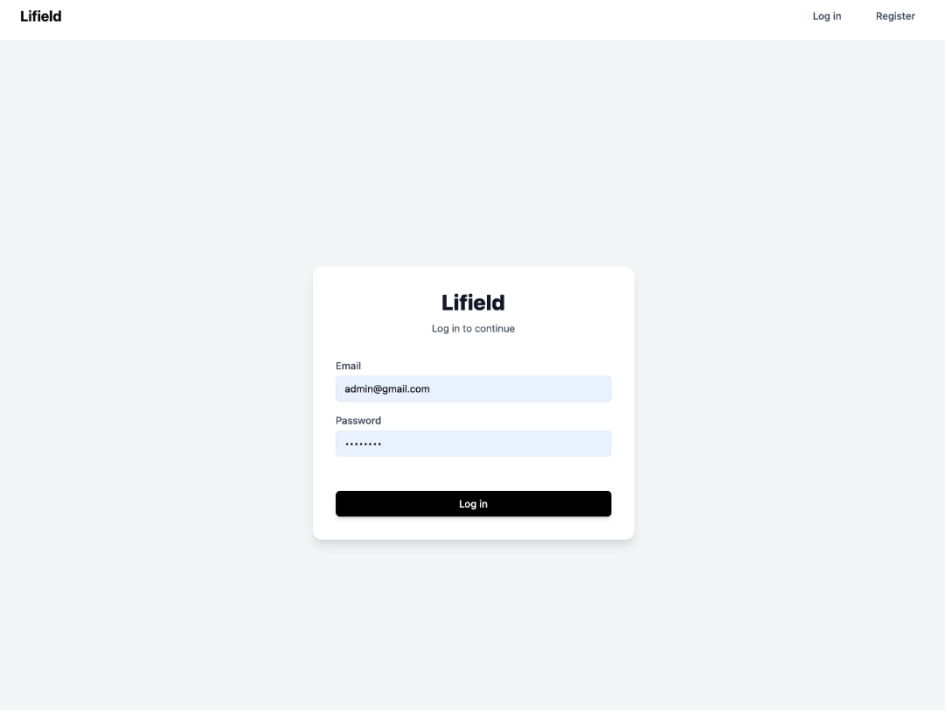

Рисунок 2.17 — Графічний інтерфейс форми автентифікації користувача в системі

Після успішної перевірки пароля за допомогою алгоритму BCrypt на стороні сервера та генерації JWT-токена, користувач автоматично перенаправляється всередину приватної частини системи. Основною сторінкою для керування користувацькими просторами виступає WorkspacePage, яка містить деревоподібну структуру вкладених розділів (Sections).

Завдяки використанню технології Apollo Client та стратегії кешування cache-and-network, перехід між розділами відбувається без візуальних затримок: система миттєво відображає дані з локального InMemory-кешу браузера, паралельно оновлюючи контент у фоновому режимі з бази даних PostgreSQL.

Основний функціонал платформи можна побачити всередині конструктора розділу при активації режиму редагування. У цьому стані інтерфейс динамічно підключає додатковий шар взаємодії: фонова область сторінки покривається SVG-паттерном сітки.

Користувач отримує можливість вільно перетягувати блоки та змінювати їхні розміри в межах 24 умовних колонок за допомогою технології Drag-and-Drop. Візуальне відображення конструктора в активному стані редагування наведено на рис. 2.18.

Для додавання нових модулів у режимі редагування використовується бічна панель інструментів SectionSidebar, яка містить перелік доступних віджетів. Особливу вагу в системі має віджет таблиці (TableWidget). При виклику універсального діалогового інтерфейсу WidgetModal для конфігурації таблиці, користувачам стає доступною дворівнева система налаштувань. У вкладці «Структура таблиці» налаштовуються метадані (додавання, видалення та іменування колонок), а у вкладці «Записи даних» — безпосереднє йде заповнення комірок інформацією.

Будь-які проміжні маніпуляції у формах за допомогою механізму чернеток зберігаються в локальному стані pendingChanges. Інтерфейс фабричного модального вікна налаштування таблиці зображено на рис. 2.19.

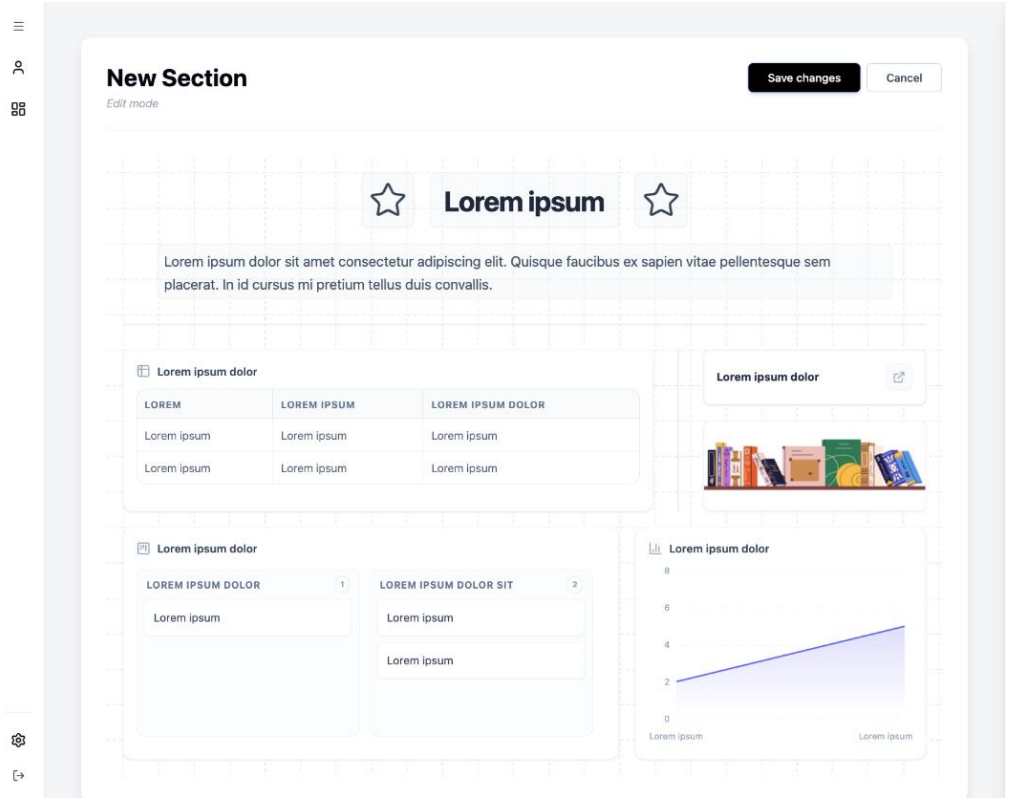

Рисунок 2.18 — Інтерфейс конструктора сторінок в активному режимі редагування

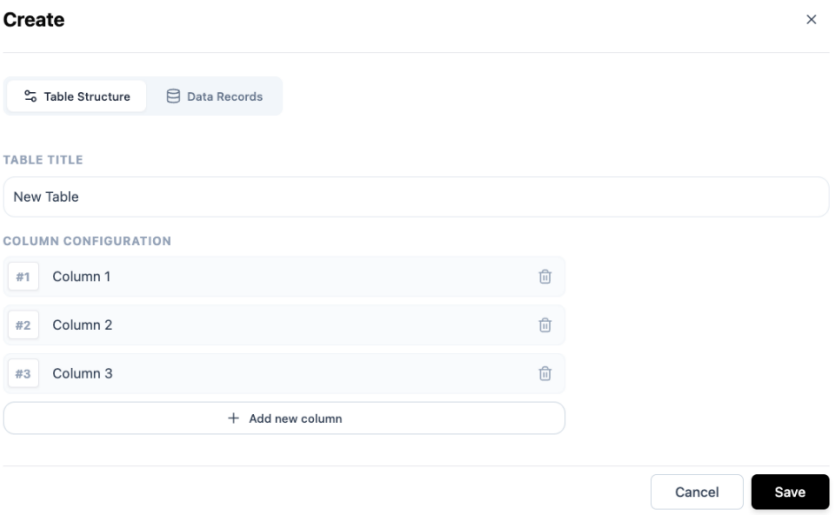

Рисунок 2.19 — Візуальний інтерфейс модального вікна конфігурації та заповнення даними віджета TableWidget

Кінцева синхронізація локальних змін із базою даних сервера відбувається атомарно при натисканні кнопки «Save Changes».

Демонстраційне тестування системи в різних браузерах (Google Chrome, Mozilla Firefox, Safari) підтвердило повну адаптивність 24-колонкової моделі, стабільність автентифікації та високу швидкість відклику інтерфейсу.

Висновки до розділу:

У даному розділі було проведено детальне проєктування та технічну реалізацію серверної та клієнтської частин вебплатформи організації персонального простору. Розглянуто особливості побудови гібридної структури бази даних СУБД PostgreSQL на основі поєднання реляційного каркаса з документо-орієнтованим типом JSONB, а також здійснено налаштування унікальних обмежень, рекурсивних зв'язків та правил каскадного видалення за допомогою Fluent API фреймворку Entity Framework Core.

Обґрунтовано та реалізовано логіку управління транзакціями за допомогою патерну Unit of Work, інтегрованого у конвеєр ASP.NET Core через спеціальне проміжне програмне забезпечення, що у зв'язці з інфраструктурним сервісом FileCleanupService забезпечує атомарність операцій та автоматичне очищення фізичних медіаресурсів на диску сервера. Розроблено шар клієнт-серверної взаємодії на базі GraphQL-сервера Hot Chocolate з декомпозицією точок входу, впровадженням декларативної авторизації, Fluent-валідації та глобального фільтра маскування внутрішніх помилок GraphQLErrorFilter.

На клієнтському рівні спроєктовано SPA-застосунок на React із гнучким інфраструктурним мостом Apollo Client, що реалізує ланцюжок посилань для автоматичної прошивки авторизаційних токенів та заголовків локалізації системи через i18next. Реалізовано інтерактивний конструктор на базі адаптивної 24-колонкової сітки з підтримкою логіки відкладеного збереження чернеток та фабрики віджетів. На кінець проведено опис компонентів графічного інтерфейсу розробленої вебплатформи.

ВИСНОВКИ

У ході дослідження було проведено комплексну оцінку ефективності сучасних архітектурних підходів до побудови систем організації персонального простору на базі технологічного стеку .NET, GraphQL з сервером Hot Chocolate та бібліотеки React у зв'язці з Apollo Client. На основі проведеного аналізу предметної області та існуючих рішень-аналогів було спроєктовано, програмно реалізовано та успішно протестовано нову платформу. Головною інженерною перевагою розробленого застосунку є впровадження адаптивної 24-колонкової модульної сітки та фабрики віджетів, що дозволяє кінцевим користувачам гнучко налаштовувати структуру, візуальне наповнення та функціональні модулі свого робочого середовища без зміни коду платформи.

Під час розробки серверної частини системи було успішно впроваджено гібридну модель збереження даних у СУБД PostgreSQL, яка поєднує строгу реляційну цілісність сутностей із гнучкістю документо-орієнтованого бінарного типу JSONB для зберігання поліморфних конфігурацій. Безпеку та атомарність операцій забезпечено за рахунок винесення логіки фіксації транзакцій у кастомне проміжне програмне забезпечення, зроблене за патерном Unit of Work, що разом з інфраструктурним сервісом FileCleanupService гарантує автоматичне очищення дискового простору сервера від заблокованих медіаресурсів після видалення елементів. Безпека приватного контуру графа базується на Stateless-автентифікації із застосуванням криптостійкого алгоритму хешування паролів та динамічної генерації токенів JWT.

Клієнтський рівень розроблено за стандартами Single Page Application (SPA), де логіка мережевої взаємодії інкапсульована в Apollo Client, що реалізує ланцюжок посилань для автоматичної прошивки авторизаційних заголовків та синхронізації мовного контексту з бібліотекою локалізації i18next. Оптимізація швидкодії конструктора простору досягнута завдяки інтеграції механізму відкладеного збереження чернеток, який накопичує проміжні стани віджетів у

локальній пам'яті браузера, що захищає серверне API від мережевого перевантаження та масованих запитів до бази даних. Результати тестування графічного інтерфейсу повністю підтвердили стабільність та адаптивність розробленої вебплатформи.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1 Мартін Р. С. Чиста архітектура. Харків : Фабула, 2019. 368 с.

2 Freeman A. Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. Apress, 2022. 1253 p.

3 .NET documentation - .NET. *Microsoft Learn: Build with answers in reach*. URL: https://learn.microsoft.com/en-us/dotnet/ (date of access: 07.06.2026).

4 AZ. Single Page Applications (SPAs) vs. Multi Page Applications (MPAs) in React: A Beginner-Friendly Guide. *DEV Community*. URL: https://dev.to/az_99f04027276612bd1dfd0f/single-page-applications-spas-vs-multi-page-applications-mpas-in-react-a-beginner-friendly-1k0c (date of access: 07.06.2026).

5 React. *React*. URL: https://react.dev/ (date of access: 07.06.2026).

6 Relational Database vs. Document Database: What's the Difference?. *EDB*. URL: https://www.enterprisedb.com/blog/relational-vs-document-database (date of access: 07.06.2026).

7 PostgreSQL 18.3 Documentation. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/index.html (date of access: 07.06.2026).

8 GraphQL vs REST: Choosing the Right API Architecture for Your Project. *Postman Blog*. URL: https://blog.postman.com/graphql-vs-rest/ (date of access: 07.06.2026).

9 Learn | GraphQL. *GraphQL | The query language for modern APIs*. URL: https://graphql.org/learn/ (date of access: 07.06.2026).

10 Getting started with GraphQL in .NET Core. *Home*. URL: https://chillicream.com/docs/hotchocolate/v16/get-started-with-graphql-in-net-core/ (date of access: 07.06.2026).

11 Documentation. *Apollo GraphQL Docs*. URL: https://www.apollographql.com/docs/ (date of access: 07.06.2026).

12 GitHub - react-grid-layout/react-grid-layout: A draggable and resizable grid layout with responsive breakpoints, for React. *GitHub*. URL: https://github.com/react-grid-layout/react-grid-layout (date of access: 07.06.2026).

13 Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10). *Microsoft Learn: Build with answers in reach*. 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 (date of access: 07.06.2026).

14 Моркун Н. В., Завсегдашня І. В. Методичні вказівки до виконання кваліфікаційної роботи бакалавру для студентів спеціальності 122 “Комп’ютерні науки”. Кривий Ріг : Видавничий центр КНУ, 2021. 50 с.

15 ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

16 ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

17 ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація).

ДОДАТОК А

Програмна реалізація логіки автентифікації та авторизації

Листинг А.1 – Програмна реалізація бізнес-логіки у сервісі AuthService

```
public class AuthService : IAuthService
{
  private const string _defaultRegisterErrorMessage = "Внутрішня помилка при реєстрації нового(-ї) користувача(-ки).";

  private readonly IUserService _userService;
  private readonly ITokenService _tokenService;

  public AuthService(IUserService userService, ITokenService tokenService)
  {
    _userService = userService;
    _tokenService = tokenService;
  }

  public async Task<AuthPayload> LoginAsync(LoginInput input)
  {
    var user = await _userService.GetByEmailAsync(input.Email);
    if (user == null || !BCrypt.Net.BCrypt.Verify(input.Password,user.PasswordHash))
    {
      throw new InvalidCredentialsException();
    }

    var token = _tokenService.GenerateToken(user);

    return new AuthPayload
    {
      User = new UserDto { Id = user.Id, Username = user.Username, Email = user.Email },
      Token = token
```

```
    };
}

public async Task<AuthPayload> RegisterAsync(RegisterInput input)
{
    if (await _userService.ExistsByEmailAsync(input.Email))
    {
        throw new EmailAlreadyExistsException();
    }

    if (await _userService.ExistsByUsernameAsync(input.Username))
    {
        throw new UsernameAlreadyTakenException();
    }

    var passwordHash = BCrypt.Net.BCrypt.HashPassword(input.Password);

    var userInput = new User
    {
        Username = input.Username,
        Email = input.Email,
        PasswordHash = passwordHash
    };

    var profileInput = new UserProfile
    {
        LastName = input.LastName ?? string.Empty,
        FirstName = input.FirstName ?? string.Empty
    };

    var settingsInput = new UserSettings
```

```
        {
            PreferredLanguage = input.PreferredLanguage?.ToEnum<Language>() ?? Language.English,
            Theme = input.Theme?.ToEnum<UiTheme>() ?? UiTheme.Light
        };

        var user = await _userService.CreateAsync(userInput, profileInput, settingsInput);
        if (user?.Id == null || user.Id == Guid.Empty)
        {
            throw new Exception(_defaultRegisterErrorMessage);
        }

        var token = _tokenService.GenerateToken(user);

        return new AuthPayload
        {
            User = new UserDto { Id = user.Id, Username = user.Username },
            Token = token
        };
    }
}
```

ДОДАТОК Б

Програмна реалізація 24-колонкової сітки

Листинг Б.1 – Алгоритм динамічного розрахунку та адаптивного рендерингу 24-колонкової сітки віджетів

```
const SectionDisplay = ({ isEditMode, layout, setLayout, sectionsData, gridConfigData, widgetsData, tableRowsData, onEditSection, onEditWidget, onUpdateKanbanTaskColumn }) => {
  const containerRef = useRef(null);
  const [width, setWidth] = useState(1200);

  const colsCount = 24;
  const dynamicMargin = width > 1024 ? 20 : width > 768 ? 14 : 8;
  const columnWidth = (width - (dynamicMargin * (colsCount - 1))) / colsCount;
  const dynamicRowHeight = columnWidth;

  useEffect(() => {
    if (!containerRef.current) return;
    const updateWidth = () => {
      if (containerRef.current) {
        const measuredWidth = containerRef.current.offsetWidth;
        if (measuredWidth > 0) setWidth(measuredWidth);
      }
    };
    const observer = new ResizeObserver(updateWidth);
    observer.observe(containerRef.current);
    updateWidth();
    return () => observer.disconnect();
  }, []);

  const applyModeSettings = (layoutArray, editMode, currentWidgets = []) => {
    return layoutArray.map(item => {
      let config = WIDGETS_GRID_CONFIG[item.type];
```

```
        if (item.type === 'divider'){
            const realWidget = currentWidgets.find(w => w.id === item.i);
            const orientation = realWidget?.content === 'vertical' ? 'vertical' : 'horizontal';
            config = WIDGETS_GRID_CONFIG.divider[orientation];
        }

        const finalConfig = config || { isResizable: false, bounds: {} };

        return {
            ...item,
            static: !editMode,
            isDraggable: editMode,
            isResizable: editMode && finalConfig.isResizable,
            ...finalConfig.bounds
        };
    });
};

const handleLayoutUpdate = (newLayout) => {
    const updated = newLayout.map(newItem => {
        const existingItem = layout.find(old => old.i === newItem.i);
        const itemType = existingItem?.type || 'unknown';

        let config = WIDGETS_GRID_CONFIG[itemType];

        if (itemType === 'divider') {
            const realWidget = widgetsData.find(w => w.id === newItem.i);
            const orientation = realWidget?.content === 'vertical' ? 'vertical' : 'horizontal';
            config = WIDGETS_GRID_CONFIG.divider[orientation];
        }
```

```
        const finalConfig = config || { isResizable: false, bounds: {} };

        return {
          ...newItem,
          type: itemType,
          static: !isEditMode,
          isDraggable: isEditMode,
          isResizable: isEditMode && finalConfig.isResizable,
          ...finalConfig.bounds
        };
      });
      setLayout(updated);
    };
```

Листинг Б.2 – Фабричний метод відтворення поліморфного контенту та генерації SVG-підкладки конструктора простору

```
const renderWidgetContent = (item) => {
    switch (item.type){
      case 'section':{
        const sectionInfo = sectionsData.find(s => s.id === item.i) || { title: 'Untitled Section' };
        return (
          <SectionCard
            section={sectionInfo}
            isEditMode={isEditMode}
            onClick={() => !isEditMode &&
navigate(`/workspace/${workspaceId}/section/${item.i}`)}
            onEditClick={() => onEditSection(sectionInfo)}
          />
        );
      }
```

```
case 'divider':{
   const widgetInfo = widgetsData.find(w => w.id === item.i) || { content: 'horizontal' };
   return (
      <DividerWidget
         content={widgetInfo.content}
         isEditMode={isEditMode}
         onEdit={() => onEditWidget(widgetInfo)}
      />
   );
}
case 'text':{
   const widgetInfo = widgetsData.find(w => w.id === item.i);
   const defaultContent = WIDGETS_GRID_CONFIG.text.defaultContent;
   let textConfig = defaultContent;

   if (widgetInfo && widgetInfo.content){
      try {
         textConfig = typeof(widgetInfo.content) === 'string'
            ? JSON.parse(widgetInfo.content)
            : widgetInfo.content;
      } catch {
         textConfig = defaultContent;
      }
   }

   return (
      <TextWidget
         config={textConfig}
         isEditMode={isEditMode}
         onEdit={() => onEditWidget(widgetInfo)}
      />
```

```
    )
}
case 'image':{
    const widgetInfo = widgetsData.find(w => w.id === item.i);
    const imageConfig = widgetInfo?.content ? { content: widgetInfo.content } : { content: '' };

    return (
        <ImageWidget
            config={imageConfig}
            isEditMode={isEditMode}
            onEdit={() => onEditWidget(widgetInfo)}
        />
    );
}
case 'icon':{
    const widgetInfo = widgetsData.find(w => w.id === item.i) || { content: 'Heart' };
    return (
        <IconWidget
            content={widgetInfo.content}
            isEditMode={isEditMode}
            onEdit={() => onEditWidget(widgetInfo)}
        />
    );
}
case 'link':{
    const widgetInfo = widgetsData.find(w => w.id === item.i);
    const defaultContent = WIDGETS_GRID_CONFIG.link.defaultContent;
    let linkConfig = defaultContent;

    if (widgetInfo && widgetInfo.content){
        try {
```

```
                linkConfig = typeof(widgetInfo.content) === 'string'
                  ? JSON.parse(widgetInfo.content)
                  : widgetInfo.content;
              } catch {
                linkConfig = defaultContent;
              }
            }

            return (
              <LinkWidget
                config={linkConfig}
                isEditMode={isEditMode}
                onEdit={() => onEditWidget(widgetInfo)}
              />
            )
          }
          case 'table':{
            const widgetInfo = widgetsData.find(w => w.id === item.i);
            const defaultContent = WIDGETS_GRID_CONFIG.table.defaultContent;
            let tableConfig = defaultContent;
if (widgetInfo && widgetInfo.content){
              try {
                tableConfig = typeof(widgetInfo.content) === 'string'
                  ? JSON.parse(widgetInfo.content)
                  : widgetInfo.content;
              } catch {
                tableConfig = defaultContent;
              }
            }

            return (
```

```
        <TableWidget
          config={tableConfig}
          rows={(tableRowsData || []).filter(r => r.widgetId === item.i)}
          isEditMode={isEditMode}
          onEdit={() => onEditWidget(widgetInfo)}
        />
      )
    }
    case 'kanban':{
      const widgetInfo = widgetsData.find(w => w.id === item.i);
      const defaultContent = WIDGETS_GRID_CONFIG.kanban.defaultContent;
      let kanbanConfig = defaultContent;

      if (widgetInfo && widgetInfo.content){
        try {
          kanbanConfig = typeof(widgetInfo.content) === 'string'
            ? JSON.parse(widgetInfo.content)
            : widgetInfo.content;
        } catch {
          kanbanConfig = defaultContent;
        }
      }

      return (
        <KanbanWidget
          config={kanbanConfig}
          rows={(tableRowsData || []).filter(r => r.widgetId === item.i)}
          isEditMode={isEditMode}
          onEdit={() => onEditWidget(widgetInfo)}
          onUpdateTaskColumn={(taskId, targetColumn) => {
            if (onUpdateKanbanTaskColumn) {
```

```
                onUpdateKanbanTaskColumn(item.i, taskId, targetColumn);
            }
          }}
        />
    )
}
case 'chart':{
    const widgetInfo = widgetsData.find(w => w.id === item.i);
    const defaultContent = WIDGETS_GRID_CONFIG.chart.defaultContent;
    let chartConfig = defaultContent;

    if (widgetInfo && widgetInfo.content){
        try {
            chartConfig = typeof(widgetInfo.content) === 'string'
                ? JSON.parse(widgetInfo.content)
                : widgetInfo.content;
        } catch {
            chartConfig = defaultContent;
        }
    }

    return (
        <ChartWidget
            config={chartConfig}
            rows={(tableRowsData || []).filter(r => r.widgetId === item.i)}
            isEditMode={isEditMode}
            onEdit={() => onEditWidget(widgetInfo)}
        />
    )
}
```

```
        default:
            return <div className="p-4 bg-gray-100 rounded-xl">Unknown Widget</div>;
    }
  };
const gridStyle = isEditMode ? {
    backgroundColor: '#ffffff',
    backgroundImage: `url("data:image/svg+xml,%3Csvg width='${columnWidth + dynamicMargin}' height='${dynamicRowHeight + dynamicMargin}' viewBox='0 0 ${columnWidth + dynamicMargin} ${dynamicRowHeight + dynamicMargin}' xmlns='http://www.w3.org/2000/svg'%3E%3Cpath d='M ${columnWidth + dynamicMargin} 0 L 0 0 0 ${dynamicRowHeight + dynamicMargin}' fill='none' stroke='%23cbd5e1' stroke-width='1' stroke-dasharray='4,4'/%3E%3C/svg%3E")`,
    backgroundPosition: `${dynamicMargin}px ${dynamicMargin}px`,
  } : {};
```

ДОДАТОК В

Програмна реалізація форми заповнення

Листинг В.1 – Компонент десеріалізації NoSQL-конфігурації та динамічної модифікації матриці даних віджета

```
const TableForm = ({ onClose, onSave, onDelete, widget, rows = [], mode = "edit" }) => {
  const [isEditing, setIsEditing] = useState(mode === "create");
  const [title, setTitle] = useState(t(defaultContent.title));
  const [columns, setColumns] = useState([t(defaultContent.columns[0]),
t(defaultContent.columns[1]), t(defaultContent.columns[2])]);
  const maxColumns = columns.length;

  const parseRowCells = (rowCells) => {
    if (!rowCells) return [];
    if (typeof rowCells === 'string') {
      try {
        return JSON.parse(rowCells);
      } catch (err) {
        return [];
      }
    }
    return rowCells;
  };

  const prepareRows = (rawRows) => {
    return [...(rawRows || [])]
      .map(row => ({
        ...row,
        cells: parseRowCells(row.cells)
      }))
      .sort((a, b) => a.rowNumber - b.rowNumber);
  };
```

```
  const [localRows, setLocalRows] = useState(() => prepareRows(rows));

  const parseWidgetContent = (contentStr) => {
    try {
      if (!contentStr) return defaultContent;
      return typeof contentStr === 'string' ? JSON.parse(contentStr) : contentStr;
    } catch {
      return { ...defaultContent, title: contentStr || defaultContent.title };
    }
  };

  useEffect(() => {
    if (widget) {
      const config = parseWidgetContent(widget.content);
      setTitle(config.title || t(defaultContent.title));
      setColumns(config.columns || [t(defaultContent.columns[0]), t(defaultContent.columns[1]),
t(defaultContent.columns[2])]);
      setIsEditing(mode === "create");
      setActiveTab(mode === "create" ? "structure" : "data");

      setLocalRows(prepareRows(rows));
    }
  }, [widget, mode, rows]);

  const handleCellChange = (rowIndex, colIndex, value) => {
    const updated = [...localRows];
    if (!updated[rowIndex].cells) updated[rowIndex].cells = [];
    updated[rowIndex].cells[colIndex] = value;
    setLocalRows(updated);
  };
```

Листинг В.2 – Алгоритми каскадної перебудови геометрії матриці таблиці та серіалізації чернетки для GraphQL-транспорту

```
  const handleAddRow = () => {
    const newRowNumber = localRows.length > 0
      ? Math.max(...localRows.map(r => r.rowNumber)) + 1
      : 1;
    const newRow = {
      id: `temp-${Date.now()}-${Math.random()}`,
      widgetId: widget.id,
      rowNumber: newRowNumber,
      cells: Array(maxColumns).fill('')
    };
    setLocalRows([...localRows, newRow]);
  };
  const handleRemoveRow = (index) => {
    const updated = localRows.filter((_, i) => i !== index);
    const reindexed = updated.map((row, i) => ({
      ...row,
      rowNumber: i + 1
    }));
    setLocalRows(reindexed);
  };
const handleAddColumn = () => {
     const newColName = `Колонка ${columns.length + 1}`;
     setColumns([...columns, newColName]);

     const updatedRows = localRows.map(row => ({
       ...row,
       cells: [...(row.cells || []), '']
     }));
```

```
    setLocalRows(updatedRows);
};
const handleRemoveColumn = (colIndex) => {
    if (columns.length <= 1) {
        toast.error(t('tableWidget.alertMinColumns'));
        return;
    }

    const updatedCols = columns.filter((_, i) => i !== colIndex);
    setColumns(updatedCols);

    const updatedRows = localRows.map(row => {
        const updatedCells = [...(row.cells || [])];
        updatedCells.splice(colIndex, 1);
        return { ...row, cells: updatedCells };
    });
    setLocalRows(updatedRows);
};
const handleSave = () => {
    const stringifiedContent = JSON.stringify({
        title,
        columns
    });

    onSave({ id: widget?.id, content: stringifiedContent }, localRows);
};
```