Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ЗАСТОСУНОК НА BLAZOR ДЛЯ ПОБУДОВИ ГРАФІКІВ ТА ВІЗУАЛІЗАЦІЇ МАТЕМАТИЧНИХ ОБЧИСЛЕНЬ

BLAZOR APPLICATION FOR GRAPHING AND VISUALIZING MATHEMATICAL CALCULATIONS

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконала: здобувачка вищої освіти
групи КН-42
Видрик Юлія Сергіївна

_____
(підпис)

Керівник: доцент
Здолбіцька Ніна Василівна

_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2026 р.
Гарант освітньої програми:
д.пед.н., професор
Тулашвілі Юрій Йосипович

_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«16» січня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ПЕРШОГО (БАКАЛАВРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Видрик Юлія Сергіївна
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Застосунок на Blazor для побудови графіків та візуалізації математичних обчислень»
Керівник доцент Здолбіцька Ніна Василівна,
затверджені наказом закладу вищої освіти від «*16*» *січня 2026* р. № 32/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«02» червня 2026 р.*
3. Вихідні дані до роботи: наукові праці відносно чисельних методів, технологічні стеки та бібліотеки для побудови кросплатформних застосунків, шаблони побудови комплексних програмних систем, технології для двомовної інтеграції, реляційні бази даних, безпечний обмін даними між сервісами, мануальне тестування працездатної інфраструктури
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проєктування, опис функціонального наповнення об'єкта проєктування, розробка й обґрунтування системного наповнення, оцінка ергономічних та надійнісних параметрів проєктованої системи, функціонально-структурна схема роботи об'єкта проєктування
5. Перелік графічного матеріалу: 22 рисунки, 19 лістингів, 2 формули, презентація.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Здолбіцька Н. В.*		
Специфікація вимог до розроблюваної системи	*Здолбіцька Н. В.*		
Розробка інфраструктури для виконання обрахунків чисельними методами	*Здолбіцька Н. В.*		
Ергономічна оцінка функціональних компонентів програмного продукту	*Здолбіцька Н. В.*		
Показник запозичень тексту	_____ %		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Сачук В. О.*		
Гарант ОПП	*Тулашвілі Ю. Й.*		

7. Дата видачі завдання *«16» січня 2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 17.02.2026 р*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 28.02.2025 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*до 12.03.2025 р*	
4	*Описати засоби розробки об'єкта проектування*	*до 26.03.2026 р.*	
5	*Практична реалізація об'єкта проектування*	*до 30.04.2026 р.*	
6	*Провести спеціальні розрахунки*	*до 18.05.2026 р.*	
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*до 02.06.2026 р.*	

Здобувачка вищої освіти _____ Юлія ВИДРИК

Керівник роботи _____ Ніна ЗДОЛБІЦЬКА

АНОТАЦІЯ

Видрик Ю. С. Застосунок на Blazor для побудови графіків та візуалізації математичних обчислень. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

В першому розділі було здійснено аналіз поточного стану предметної області, ознайомлення з існуючими аналогами та вибір методів рішення проблеми, пов'язаної з малою кількість математичних ПЗ, що здійснюють обрахунки з використанням чисельних методів. Завершальним етапом був опис завдань, які треба виконати задля досягнення мети кваліфікаційної роботи.

В другому розділі обґрунтовувався вибір технологічного стеку та засобів для вирішення поставлених завдань, описано вимоги до системи: функціональні, що розділяються на обчислювальні та користувацькі, та нефункціональні. В розділі описано архітектуру компонентів та їхню взаємодію в системі, опис сутностей, що фігуруватимуть в API користувачів.

Третій розділ містить опис розробки таких підсистем: API для обчислень, користувацьке API, RCL та її успадковувачі: вебсервіс та кросплатформний застосунок. Завершується описом процесу тестування та його результатів, і розгортанням інфраструктури на середовищах.

В четвертому розділі здійснюється ергономічна оцінка програмного комплексу – зручність використання платформи та робота основного функціоналу. У висновках знаходиться узагальнена інформація по побудованій системі, її структурі, можливих проблемах та покращеннях.

Ключові слова: чисельні методи, .NET, Blazor, ASP.NET, кросплатформність, CSnakes, візуалізація даних.

АНОТАЦІЯ

Yuliia Vydryk. Blazor application for graphing and visualizing mathematical calculations. Manuscript.

Bachelor’s qualification paper in the study program "Computer Science". Lutsk National Technical University. Lutsk, 2026.

The bachelor’s qualification paper consists of an introduction, four chapters, conclusions, a list of references and appendices.

The first chapter analyses the current state of the subject area, reviews existing alternatives and selects methods to address the problem associated with the limited number of mathematical software packages that perform calculations using numerical methods. The final stage describes the tasks to be completed in order to achieve the thesis's objective.

The second chapter justifies the choice of technology stack and tools for addressing the set tasks, and describes the system requirements: functional requirements – divided into computational and user-related requirements – and non-functional requirements. The chapter outlines the architecture of the components and their interaction within the system, as well as a description of the entities that will feature in the user API.

The third chapter contains a description of the development of the following subsystems: the computational API, the user API, RCL and its derivatives: the web service and the cross-platform application. It concludes with a description of the testing process and its results, and the deployment of the infrastructure across environments.

The fourth chapter provides an ergonomic assessment of the software suite – the platform’s usability and the operation of its core functionality. The conclusions contain summarised information on the developed system, its structure, potential issues and areas for improvement.

Keywords: numerical analysis, .NET, Blazor, ASP.NET, cross-platform, CSnakes, data visualisation.

ЗМІСТ

ВСТУП

Вирішення математичних задач може вимагати використання алгоритмів різної природи, зокрема аналітичних, що базуються на пошуку формули для розв’язання, символьних з їхніми маніпуляціями змінними або функціями та графічних, які вимагають побудову графіків. Проте в умовах інженерної реальності такі методи часто є слабкими та тяжкими: змінних може бути чимало, функція для опису явища може бути геть відсутньою (як у випадку з датчиками, які зчитують дані в умовах нестабільного навколишнього середовища). Графічні методи хоч і надають можливість описати функцію по дискретному наборі точок, проте вони також не допоможуть, якщо функція має більше трьох параметрів, або результат неможливо точно оцінити візуально. В такому випадку часто застосовують чисельні методи – вид обчислень, логіка яких ґрунтується на ітераційному виконанні певної операції, що з кожним наступним кроком наближає до розв’язку задачі.

Чисельні методи хоч і часто вивчаються студентами ІТ-спеціальностей, проте наразі в світі є небагато програмних продуктів, які могли б використовувати даний вид при обрахунках, часто вирішуючи задачу або аналітичним, або символьним методом. При цьому ті завдання, які все ж таки можливо було вирішити тільки з використанням чисельних алгоритмів, повністю ігноруються системами.

Враховуючи описану вище проблему, було сформовано таку мету даної кваліфікаційної роботи: створення системи з програмних компонентів, якої було б достатньо для вирішення типових математичних завдань: інтерполяція або інтегрування функції, розв’язок систем рівнянь різних порядків, пошук нуля функції тощо, з можливістю застосування чисельних методів.

Об’єктом даного дослідження є процес побудови системи із сервісно-орієнтованою архітектурою (SOA) та розробки кросплатформних застосунків, які б в сукупності змогли б забезпечити користувачам вільний доступ до математичних обрахунків з використанням чисельних методів.

Предметом дослідження є архітектурні рішення та технологічний стек, що дозволять розробити систему з необхідною серверною інфраструктурою та спроєктувати кросплатформну UI-систему для доступу до основного функціоналу інфраструктури.

Завдання, які будуть виконані в межах даної кваліфікаційної роботи:

1) аналіз проблеми та існуючих аналогів-застосунків на ринку;

2) вибір інструментів для розробки кросплатформного застосунку та побудови бекенд-частини в межах одного технологічного стеку;

3) пропрацювання архітектурного рішення, що описуватиме загальну структуру продукту, його програмні частини та поведінку;

4) кодова імплементація серверної інфраструктури з використанням обраної технології;

5) створення кросплатформного застосунку, з попереднім продумуванням дизайн-системи та структури відповідних UI-проєктів;

6) тестування реалізованого функціоналу та внесення правок в разі зафіксованих недоліків або розбіжностей;

7) налаштування автоматизованого розгортання всієї інфраструктури з використанням хмарних рішень, підготовка інсталяційних пакетів для встановлення застосунку на Android та Windows ОС.

Апробація. Основну проблематику та результати дослідження, проведених в межах виконання кваліфікаційної роботи, було представлено на X Міжнародній студентській науковій конференції «Концепт науки XXI: стратегії, методи та наукові інструменти» (додаток А), тези опубліковані в збірнику до конференції [1] (додаток Б).

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз сучасного стану проблеми

Математика – одна з небагатьох наук, яка має важливе практичне значення при вирішенні багатьох проблем в житті. Тісне переплетення її з іншими сферами дозволяє розширювати межі як самої математики, так і інших сфер. Чимало інженерних систем функціонують на базі математики, а її застосування можна побачити як в гастрономічній сфері та будівельній інженерії, так і в розробці програмного та апаратного забезпечення.

З розвитком історії людства, математика, як наука, також пройшла багатьох етапів еволюції. Чимало нових поставлених проблем потребували продумування та реалізації нових рішень. Спочатку людство відкрило для себе арифметику та геометрію, з врахуванням того, що потрібно було виконувати базові обрахунки та визначати властивості предметів різних геометричних умов. Зі своїм прогресом, людство зрозуміло, що можна спробувати багато речей описати певною формулою, щоб пізніше можна було легко визначити результат з підстановкою вхідних параметрів. Хоча й багато законів таким чином було віднайдено, проте крім цього було усвідомлено, що не всі речі дійсно можна описати так точно, як хотілося б. В такий спосіб людство змогло розвинути одне з відгалужень математичної сфери – чисельні методи, які базуються саме на віднаходженні наближеного розв’язку, з певною дозволеною похибкою.

Чисельні алгоритми не є такими сучасними, як могло здатися на перший погляд: прикладом цього представляється єгипетський папірус Райнда, що містить опис деяких математичних проблем, включно з описом чисельного методу пошуку кореня рівняння [2, 3]. Однак, основна проблема чисельних методів на той момент полягала в тому, що даний тип алгоритмів зазвичай вимагає ітераційного підходу та правильного визначення вхідних даних, що впливає на час, необхідний для пошуку наближеного розв’язку. З активним розвитком електронно-обчислювальних машин, науковці змогли зменшити собі

тягар постійних однотипних обрахунків, що в свою чергу тільки активніше посприяло використанню чисельних методів. Наразі, з широким доступом до потужних процесорів, дисципліна часто використовується в моделюванні прогнозів, розвитку нейромереж, інженерному проєктуванні тощо [4].

Широке використання чисельних методів в сьогоденні вимагає підготовки спеціалістів, які б потенційно могли використовувати ці алгоритми в своїй праці за спеціальністю. Зазвичай це стосується майбутніх:

– IT-спеціалістів (розробка програмного забезпечення може бути різноманітною та вимагати різних алгоритмів рішення);

– прикладних математиків та фізиків (різні проблеми можуть вимагати різних рішень);

– інженерів (під час розробки можна спробувати змоделювати різні ситуації та оптимізувати побудову моделей-прототипів);

– аналітиків (передбачення майбутніх тенденцій на основі наявних даних).

В такому випадку вищі заклади освіти впроваджують вивчення окремої дисципліни, яка дозволить базово ознайомити студентів, зазначених вище спеціальностей, з базовими та загальновідомими чисельними алгоритмами. Типово у курс входять методи, які дозволяють:

– шукати корені рівнянь;

– розв’язувати системи лінійних та нелінійних алгебраїчних рівнянь;

– виконувати диференціювання;

– інтегрувати, інтерполювати та оптимізувати функції.

У випадку студентів факультетів інформаційних технологій часто, до теоретичної частини, в лекціях наводяться блок-схеми до відповідно розглянутих алгоритмів, і також кодова реалізація з використанням таких мов програмування, як Python і MATLAB. У випадку з лабораторними студенти мають самостійно реалізовувати програмно чисельні методи та, за можливості, візуалізувати результати з використанням пакетів для побудови графіків [5]. Проблематика в таких завданнях постає в тому, що:

– неможливо швидко перевірити чи дійсно результати дійсні для вказаних

вхідних параметрів;

– тяжко швидко згадати, як саме працює певний чисельний метод покроково. Хоча й завжди можна звернутися до теоретичного матеріалу, але це може бути не завжди зручно, доступно і зрозуміло.

Є чимало застосунків для обрахунків, що орієнтовані на зручне візуальне доповнення. Проте дуже мало є тих, які орієнтовані саме на чисельні методи, а не аналітичні. Відповідно можна виокремити звідси неможливість перевірити якість обрахунків з використанням чисельних методів, що вимагає пошуку вирішення проблеми.

На основі вище сказаного, можна зрозуміти, що одним з найкращих запропонованих рішень може бути створення зручного для користувача застосунку, який дозволятиме проводити обрахунки з використанням чисельних алгоритмів.

1.2 Аналіз аналогічних програм, баз даних, систем, обґрунтування вибраного напряму та вибір методів рішення

Наразі є чимало різних застосунків, які дозволяють як школярам, так і студентам розв’язувати типові математичні задачі: арифметичні обрахунки, пошуки розв’язків рівнянь або їх систем, побудова графіків тощо. Багато з них було проаналізовано задля опису майбутнього функціоналу системи.

Photomath – мобільний застосунок, що доступний як для ОС Android, так і для iOS. Є одним з найпопулярніших застосунків серед українських здобувачів освіти, завдяки своєму функціоналу із поетапним поясненням кроків та можливістю швидко сфотографувати певний математичний вираз, що відразу буде надісланий на обробку після підтвердження. У випадку внесення даних вручну, надається багатофункціональна система введення даних, що дозволяє зручно з мобільного пристрою вводити як і складні формули, так і матриці. Після вирішення завдання, програмне забезпечення відображає рішення в багатьох варіантах (наприклад, символьному або графічному) та може покроково

пояснити сам процес віднаходження розв’язку. Останній функціонал високо цінується в освітній сфері, адже він може розвивати в користувачів прагнення до пошуку не самого готового рішення проблеми, а намагання зрозуміти яким чином розв’язок було отримано. З недоліків системи можна виокремити те, що система не пропонує різні методи вирішення однієї поставленої задачі. Якщо розглядати в розрізі з чисельними методами, система надає можливість вирішення тільки тих завдань, які можуть потенційно бути обчислені аналітично [6]. Неможливість якось зберігати введені дані для вирішення задач також ускладнює користування застосунком, особливо якщо проводяться якісь експериментальні дослідження.

WolframAlpha – програмне забезпечення, яке представлене як у вебверсії, так і в мобільному додатку, розроблене інженерами, що створили таку мову програмування, як Wolfram Language. Оскільки дана мова базується на символьних обчисленнях, то й сама система також використовує в своїй механіці даний вид обрахунків (в поєднанні з чисельними методами). В порівнянні з Photomath, WolframAlpha підтримує вирішення задач чисельними алгоритмами. Проте, з метою використання якомога менших ресурсів машини, прикладна програма не дозволяє гнучко налаштовувати вхідні параметри для розв’язку проблеми, зокрема: допустиму похибка, допустиму максимальну кількість ітерацій тощо [7]. З таким підходом користувачі також не можуть порівнювати результати виконання різних методів, оскільки не відображаються такі показники, як кількість виконаних ітерацій, час виконання алгоритму ітд. Відсутність пояснення виконаних кроків для багатьох задач впливає на прозорість самих обрахунків.

Окрім іноземних продуктів можна згадати вебплатформу mathros – український сайт, що надає доступ до немалої кількості інформації про різні аналітичні та чисельні алгоритми. Майже кожна стаття містить коротку теоретичну відомість, блок-схеми з описом самих алгоритмів та практичні завдання для розв’язку. Також на платформі можна віднайти інструменти-калькулятори для обрахунків дробів, похідних, матриць тощо. Хоча вони й представлені переважно для пошуку розв’язку СЛАР, інтегралів та

диференціальних рівнянь, проте вони також містять покроковий опис виконання завдань. Відсутність можливості гнучкого введення вище згаданих параметрів та порівняння методів також є недоліками вебсайту [8].

Такі прикладні системи, як Geogebra та Desmos добре спеціалізуються саме на графічних методах обрахунків, що відрізняються за своєю суттю від чисельних методів. Найкраще розглядати чисельні алгоритми в порівнянні з аналітичними та символьними видами.

Отже, задля уникнення вище описаних проблем, розроблюване програмне забезпечення, по-перше, має забезпечувати користувачу можливість зручно вводити вхідні параметри. Основний функціонал має бути зосереджений на використанні безпосередньо чисельних алгоритмів, оскільки розглянуті вище системи зосереджені переважно на аналітичних, символьних та графічних методах. Якісний досвід споживача продукту має бути забезпечений безперебійною роботою системи, оскільки йому надається чимало «свободи» в плані введення даних. Користувацький інтерфейс має бути зрозумілим та інтуїтивним, і, що не менш важливо, розв’язок задачі має відображатися з додатковими показниками: затрата часу на виконання алгоритму, кількість задіяних ітерацій та чи було досягнуто необхідної точності без перевищення показника максимально допустимих циклів методу. З врахуванням спільних позитивних моментів у всіх розглянутих застосунках, система має бути зосереджена більше не на тому, щоб показати моментальне рішення проблеми, а на тому, щоб спонукати користувача зрозуміти чому певні методи швидше або точніше відпрацювали з такими-то введеними даними. Відображення списку з роз’ясненими етапами в кожному алгоритмі якраз допоможе з цією задачею. Також важливо впровадити й саму можливість перевикористання введених даних, чого дійсно бракувало у всіх застосунках вище. Для широкого вибору можливості доступу до системи, варто попіклуватися про її кросплатформність, що включає в себе розробку вебдодатку, десктопного та мобільного варіантів застосунків.

1.3 Аналіз і вибір засобів проектування кросплатформних застосунків та серверних інфраструктур

Кросплатформний застосунок – програмне забезпечення, яке здатне безперебійно працювати незалежно від операційної системи на основі однієї кодової бази. Зазвичай такий тип систем розробляють, якщо потрібно швидко та належно розробити один застосунок одночасно як і для вебу, так і для нативних пристроїв. У випадку якоїсь очікуваної різниці в поведінці в залежності від системи, в якому має виконуватися програмне забезпечення, мабуть, кращим рішенням буде розробляти окремі додатки (так звана нативна розробка). Під час виконання завдання даної кваліфікаційної роботи достатньо буде мати одну кодову базу для усіх необхідних платформ.

Серед найпопулярніших фреймворків для кросплатформної розробки виокремлюються наступні екземпляри:

– Flutter. Популярний інструмент для побудови мобільних додатків, створений компанією Google. Користування даним фреймворком вимагає знань в такій мові програмування, як Dart. З корисного функціоналу можна виділити «Hot Reload» (дозволяє відображати внесені зміни в коді без необхідності перекомпіляції застосунку) та багату бібліотеку з віджетами, що є частиною Material Design. Дизайн-система також розроблена Google та акцентується на «плоскості» з використанням тіней та шарів, контрастності елементів тощо;

– React Native. Фреймворк, який є open-source проєктом від компанії Meta Platforms. З ним розробники, що звикли працювати з JavaScript, можуть також швидко навчитися вибудовувати власні кросплатформні застосунки. Зі зручностей тут також є можливість бачити зміни в коді без перекомпіляції. Flipper – платформа, що постачається в комплекті з React Native, дозволяє інспектувати як структуру UI-компонентів, так і стан мережі, баз даних тощо;

– .NET MAUI (Multi-platform App UI). Кросплатформне рішення, що є частиною .NET-екосистеми та «наслідуванням» від Xamarin. Маючи так само функцію «Hot Reload» та можливість використання єдиної кодової бази,

написаної з використанням таких мов, як C# та XAML, система пропонує кросплатформні API для отримання доступу до даних відносно батереї, мережі, GPS тощо [9].

Зважаючи на вище перелічені фреймворки, для створення застосунку було обрано використання .NET MAUI, що відкриває шляхи для роботи безпосередньо з .NET. Вибір даного стеку базувався на:

– особистому досвіді роботи автора з даною екосистемою, який можна вважати результативним;

– активній підтримці екосистеми з боку власника продукту – Microsoft – та постійне вдосконалення платформи;

– можливості використовувати стек як для побудови UI-проєктів, так і побудови бекенд-сервісів.

C# – об’єктно-орієнтована мова програмування, що є основою даного технологічного стеку. Характерна їй строга типізація, асинхронне програмування (обгорнуте, як синтаксичний «цукор» для зручності) та LINQ (функціонал для роботи з даними) дозволяють оволодіти мовою досить швидко та не дуже турбуватися про потенційні моменти з пам’яттю (оскільки вбудований Garbage Collector займається керуванням очищення ресурсів) [10].

Для побудови REST API буде використовуватися ASP.NET Core – ще один високопродуктивний фреймворк, що призначений для створення API, мікросервісів тощо. Кросплатформність та підтримка Dependency Injection робить його хорошим інструментом для розбудови майбутньої системи. Величезна користувацька спільнота показує, що технологія є популярною, а отже і підтримуваною зі сторони Microsoft [11]. Для зберігання даних користувачів було вирішено обрати реляційну модель, реалізовану за допомогою такої системи управління базами даних, як MS SQL Server. Дана СУБД базується на Transact-SQL – розширенні SQL, що містить більш розширений функціонал, пропрацьований спеціалістами Microsoft та Sybase [12].

Таким чином, всі вище згадані інструменти будуть використані для швидкої побудови кросплатформного застосунку та серверної інфраструктури.

1.4 Постановка завдання на кваліфікаційну роботу бакалавра

Досягнення основної мети кваліфікаційної роботи – реалізація програмної інфраструктури, що забезпечить можливість користувачам проводити ряд обрахунків з використанням чисельних методів – вимагає виконання наступних завдань:

1) аналіз проблеми та існуючих аналогів-застосунків на ринку;

2) вибір інструментів для розробки кросплатформного застосунку та побудови бекенд-частини в межах одного технологічного стеку;

3) пропрацювання архітектурного рішення, що описуватиме загальну структуру продукту, його програмні частини та поведінку;

4) кодова імплементація серверної інфраструктури з використанням обраної технології;

5) створення кросплатформного застосунку, з попереднім продумуванням дизайн-системи та структури відповідних UI-проєктів;

6) тестування реалізованого функціоналу та внесення правок в разі зафіксованих недоліків або розбіжностей;

7) налаштування автоматизованого розгортання всієї інфраструктури з використанням хмарних рішень, підготовка інсталяційних пакетів для встановлення застосунку на Android та Windows ОС.

Повне завершення даних завдань знаменуватиме успішне досягнення поставленої мети в межах кваліфікаційної роботи.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання

.NET – екосистема, що містить велику кількість програмних пакетів в своїй NuGet-базі, створених іншими спеціалістами. Проте незважаючи навіть на ту кількість пакетів, виникає проблема із побудовою таких інфраструктур, що можуть вимагати математичні обрахунки різного роду: обчислення похідних, диференціювання тощо. Деяка частина з них просто не оновлюється та не підтримується вже багато часу, інша – містить недостатній функціонал. Відповідно, було прийнято рішення під час побудови системи інтегрувати Python-інфраструктуру – найбагатшу наукову екосистему, яка широко використовується в області виконання обчислень.

Є декілька можливих NuGet-пакетів, які можуть допомогти у включенні: Python.NET, CSnakes, IronPython. Обраний пакет має дозволити на рівні побудованих C#-сервісів виконувати обрахунки з використанням типових математичних Python-пакетів: SymPy, NumPy тощо. IronPython не дозволяє використовувати вищезгадані бібліотеки, оскільки доступ до останніх реалізований через CPython C API – програмне забезпечення, що є придатним тільки для стандартного CPython, який не використовується IronPython [13]. Всі інші пакети вимагають використання dynamic-об’єктів, що є менш типобезпечно. Зважаючи на це, було вирішено обрати такий пакет, як CSnakes, оскільки той має автоматичну генерацію інтерфейсів на основі написаної кодової бази з використанням Python. Нюансом в цій бібліотеці є обмеження у формі найменування файлів скриптів та папок: створення інтерфейсів та методів базується якраз на назвах файлів та функцій, тому усі найменування мають бути унікальними та написаними в стилі Snake Case [14].

Нижче перераховано інші бібліотеки, що будуть застосовані під час розробки серверної інфраструктури:

– Entity Framework Core. ORM, що дозволить маніпулювати даними з бази про користувачів без необхідності написання SQL-скриптів. Натомість тут використовується звичний для C#-спеціалістів LINQ-синтаксис. Такий підхід дозволить зменшити кількість недопрацювань за рахунок типобезпеки та розширених функцій бібліотеки;

– Swashbuckle. Фреймворк, що використовуватиметься тут для впровадження можливості мануального тестування користувацького API та API з обрахунками. UI для тестування генерується під час кожного запуску проєкту на основі доданих контролерів з врахуванням нотацій та summary-коментарів;

– Serilog. Популярна бібліотека для структурованого логування, що буде використовуватися для ведення журналу функціонування системи (включно із нативними застосунками). Основною перевагою є легка та невелика за об’ємом конфігурація, яка може легко змінюватися в залежності від розроблюваної системи.

Як зазначалося в попередньому розділі, для зберігання даних користувачів буде використано реляційну базу даних, яка працюватиме під управлінням MS SQL Server. Вище згаданий інструмент – Entity Framework Core – тут виступатиме ORM-системою, що перекладатиме написані LINQ-конструкції у їхню SQL-репрезентацію.

Створення кросплатформного застосунку вимагатиме використання .NET MAUI. Оскільки проєкт має включати вебверсію, що не підтримується MAUI, можна скористатися такою технологією, як Blazor Hybrid. З нею, розроблений UI-інтерфейс в межах RCL, можна одночасно використовувати як у вебпроєкті, так і у нативних додатках з попередньо налаштованим WebView. Варто зауважити, що Blazor – фреймворк, що подібний на React, Angular ітд, проте він дозволяє замість JavaScript використовувати саме C#. Звідси, всі компоненти та сторінки, що розробляються з Blazor, послуговуються в роботі HTML (основний каркас компонентів), CSS (стилізація) та C# (мова програмування для написання кодової частини елементів). В межах Blazor Hybrid та кросплатформної розробки, Blazor зазвичай використовується під час побудови спільної бібліотеки для зберігання

UI-компонентів – Razor Class Library. Дана бібліотека дозволяє створити єдину кодову базу, що зможе бути використана в межах створеного вебпроєкту (наприклад, Blazor Web App) та створеного MAUI-проєкту. Додатковою зручністю є те, що розробникам, які працювали тільки з Razor (синтаксис, що дозволяє поєднувати C# та HTML), не доведеться додатково вивчати особливості роботи із XAML в площині MAUI [15].

В межах розробки власної UI-бібліотеки було вирішено використати наступні технології:

– PostCSS. Постпроцесор CSS, що дозволить автоматично додавати вендорні префікси задля підтримки кросбраузерності та форматувати CSS-файли у валідний формат. Останній пункт необхідний, оскільки передбачається застосування вкладеності CSS-класів при побудові структури;

– Vite. Інструмент, який надасть змогу обробляти та оптимізовувати статичні ресурси без втрати якості: мініфікація головних CSS та JS файлів, зменшення розміру зображень без втрати якості тощо. Також Vite буде застосовано для побудови SVG-спрайту та підключення завантажених шрифтів;

– Node Package Manager або npm. Менеджер пакетів, що широко використовується задля завантаження додаткових бібліотек. З ним буде надана можливість налаштувати необхідні нам конфігураційні файли (плагіни для PostCSS та безпосередньо Vite) та підключити додаткові пакети на виведення LaTeX-тексту в читабельному вигляді («katex»), введення математичних виразів («math-live»), відображення графіків («highcharts») тощо.

Для зручності по введенню математичних виразів було вирішено додати розпізнавання математичних виразів із зображення користувача. Реалізація даного завдання вимагає використання моделей, попередньо навчених на розпізнавання об’єктів. Оскільки саме завдання по розпізнаванню математичних символів та виразів є достатньо складною процедурою, було вирішено частково інтегрувати систему з Google Gemini API, яка може надати доступ до вже готових моделей. Задля передачі якомога кращих та якісніших фотографій на обробку, користувачу буде запропоновано попередній перегляд та можлива обробка файлу

з використанням Cropper.Blazor. Дана бібліотека є продуктом-обгорткою над популярним Cropper.js і дозволяє маніпулювати зображенням моментально в інтерфейсі користувача перед відправкою. Точність розпізнавання виразів та функцій буде більшою завдяки прибиранню зайвих шумів та інших об’єктів.

Отже, застосування вищезгаданих бібліотек та фреймворків, що є частиною або .NET, або npm, або Python-екосистеми, дозволить виконати поставлені завдання та досягнути основної мети кваліфікаційної роботи – побудови інфраструктури для виконання обрахунків з використанням чисельних методів.

2.2 Функціонально-структурна схема роботи об’єкта проектування

Проєктування архітектури програмного забезпечення є важливою і першочерговою частиною життєвого циклу розробки. Даний етап включає аналіз вимог поставленої задачі та безпосереднє пропрацювання майбутньої архітектури розроблюваної інфраструктури (включно з її компонентами).

Функціональні вимоги можна поділити на такі дві частини: обчислювальні та користувацькі. Обчислювальні вимоги включають:

– пошук кореня рівняння в заданих межах функції з використанням таких методів, як бісекції (дихотомії), простих ітерацій, Ньютона, січних та комбінованого методу. Останній має поєднувати переваги методів Ньютона та січних;

– розв’язання як лінійних (методи Крамера та Гауса), так і нелінійних систем рівнянь (методи простих ітерацій та Гауса-Зейделя). Допускаються до розв’язання системи не вище 4-го порядку;

– інтерполяцію функцій методами Ньютона, Лагранжа та кубічного сплайну. Має дозволятися введення як самої функції у формулі, так і у вигляді набору координат;

– чисельне диференціювання з використанням методів скінченної різниці (прямої, зворотної та центральної) або через поліном Лагранжа. Мають

підтримуватися перший та другий порядки диференціювання;

– чисельне інтегрування, що включає використання методів з прямокутниками та трапеціями, метод Сімпсона (параболи);

– оптимізацію (максимізація та мінімізація функцій). Одновимірна оптимізація має здійснюватися із застосуванням методу або рівномірного пошуку, або золотого перетину. Для багатовимірної оптимізації необхідно додатково програмно реалізувати метод градієнтного спуску/підйому;

– розв’язання звичайних диференціальних рівнянь (ЗДР) методами Ейлера (включно з удосконаленим), Рунге-Кутта (2-го та 4-го порядків) та Пікара.

Окрім описаних вище типів математичних задач, система також має підтримувати можливість порівняння методів на основі різних показників: загальний час виконання алгоритму, кількість здійснених ітерацій тощо.

Користувацькі вимоги передбачають:

– реалізацію інтерактивних графіків для візуалізації досліджуваної функції або масиву даних;

– можливість перегляду виконаного алгоритму у покроковому вигляді;

– експорт отриманих результатів у форматі PDF-файлу. Файл має містити введені користувачем дані, покрокове вирішення задачі та результати виконання алгоритму;

– двомовний користувацький інтерфейс: англійська та українська мови;

– отримання текстового значення функції із зображення з використанням OCR-моделі від Google;

– створення інформаційних модальних вікон, що дозволять дізнатися про кожну задачу більше інформації.

Для зареєстрованих користувачів має бути надана можливість:

– здійснювати реєстрацію та авторизацію в системі;

– маніпулювати збереженими користувацькими даними (ім’я, електронна пошта, пароль) включно з видаленням зареєстрованого облікового запису;

– відновити пароль на випадок, якщо користувач його забув. При спробі його відновлення має прийти електронний лист на вказану під час реєстрації

пошту із посиланням на сторінку заміну паролю;

– збереження історії обчислень в межах 100 записів на одного користувача;

– збереження вхідних даних, введених користувачем, включно з можливістю їхнього майбутнього перевикористання;

– збереження останніх експортованих результатів в межах 10 записів на одного користувача. UI має дозволяти завантажити результати повторно, без необхідності повторного введення даних та виконання алгоритмів.

UI-сервіс має бути реалізований у форматі вебверсії та нативного застосунку (під Windows та Android). Чутливі користувацькі дані (пароль або токен) мають зберігатися в зашифрованому вигляді. Будь-якому користувачу має бути дозволено переглядати тільки свої дані.

Наведені нижче архітектурні діаграми побудовані із застосуванням моделі C4 – техніки, що дозволяє представляти архітектуру систем у формі контейнерів та компонентів [16]. Якщо контейнери представляються незалежними підсистемами одного програмного комплексу, то компоненти є їхніми частинами. Під час проєктування архітектури було розроблено 3 рівні ієрархії:

1) контекстний;

2) контейнерний;

3) компонентний.

Перший рівень відображатиме загальний стан системи та взаємодію із зовнішніми сервісами. Другий рівень дозволить побачити систему в розрізі можливих контейнерів: застосунків та внутрішніх сервісів системи. Третій рівень дозволить відобразити компоненти таких контейнерів, як Calculation API та User API. Четвертий рівень діаграми за C4 – рівень коду – не є обов’язковим до створення, оскільки він є часто змінюваним під час життєвого циклу ПЗ.

Зважаючи на зазначені вище вимоги було спроєктовано архітектуру, контекстний рівень якої відображено на рисунку В.1 в додатку В, а контейнерний рівень – на рисунку В.2 в додатку В. Ця архітектура є сервісно-орієнтованою (SOA, Service-Oriented Architecture), оскільки наші сервіси можуть існувати цілком незалежно одне від одного. За взаємодією між компонентами, програмний

комплекс побудований за принципом «клієнт-сервер», де клієнтом виступатимуть UI-сервіси (вебсервіс та нативні застосунки), а серверами – сервіс для обрахунків (Calculation API) та сервіс для керування користувачами (User API). Для комунікації із серверною інфраструктурою з використанням HTTP-протоколу, UI-сервіси матимуть окремі реалізовані HTTP-сервіси.

Також можна побачити, що серверна інфраструктура також звертається до інших зовнішніх сервісів: Google Gemini API (надає модель для того, аби можна було розпізнати математичний вираз із певного зображення) та Gmail SMTP Server (надсилання електронних листів). Якщо перший за своєю суттю є REST API, до якого можна звертатися також з використанням HTTP-протоколу, то останній вимагає SMTP-протокол. Всі необхідні credential-дані, які будуть використовуватися для доступу до зовнішніх сервісів, зберігатимуться у відповідних .env-файлах (під час локальної розробки) та environment змінних в межах або GitHub, або Azure, платформ (для майбутніх розгорнених систем). Таким чином буде знижено шанс того, що такі чутливі дані будуть скомпрометовані та використані в інакших, аніж передбачено, цілях.

Розглянемо побудовану архітектуру для сервісу обрахунків з допомогою компонентної діаграми, відображеної на рисунку В.3 в додатку В. Точкою входу в дану підсистему є Calculation Controllers, що будуть імплементовані як ASP.NET Core MVC Controllers. Всі контролери впроваджуватимуть ті створені BLL-сервіси, які їм будуть необхідні. На кожне окреме завдання (пошук кореня, інтерполяція ітд.) буде створюватися окремий контролер та окремий сервіс для обчислень задля дотримання принципу єдиної відповідальності (Single Responsibility Principle). Всі сервіси бізнес-логіки, що відповідатимуть за обрахунки, будуть використовувати інтерфейси, створені CSnakes на основі написаних Python-скриптів. Кожен такий скрипт відповідатиме за свій тип обрахунку (наприклад, інтегрування з використанням методу трапецій).

В межах Calculation API буде також створено окремий контролер, сервіс та провайдер задля можливості надсилання файлу-зображення на обробку до необхідної OCR-моделі з Google-інфраструктури. Це вимагатиме створення

додаткового інтерфейсу, який буде імплементований кожною допустимою реалізацією (до прикладу, одна реалізація надсилатиме запит до Google Gemini API, інша – до іншого сервісу). Таким чином буде застосовано такий паттерн проєктування, як «Стратегія», що дозволить в майбутньому логіку з надсиланням запитів не змінювати, а просто розширити ще однією «стратегією». Варто зауважити, що Calculation API можна реалізувати у формі багатошарової архітектури, де кожен шар відповідає за свій функціонал:

– презентаційний шар у формі контролерів;

– функціональний шар у формі сервісів бізнес-логіки;

– шар міжмовної інтеграції для виконання Python-скриптів.

Варто ознайомитися також із загальним процесом виконання обрахунку від моменту введення початкових параметрів користувачем і до моменту відображення результатів виконання запиту, на основі схеми, зазначеної в рисунку В.4 в додатку В.

Ця схема є sequence-діаграмою, основна мета якої і є візуалізація процесів. Загальний процес виконання обрахунків можна описати в такі етапи:

– заповнення користувачем необхідних полів в залежності від математичної задачі;

– натискання на кнопку «Обрахувати», що в свою чергу викликає відповідний метод з HTTP-сервісом;

– вище згаданий метод формує тіло запиту та, передавши необхідні параметри, надсилає запит на відповідну кінцеву точку контролера (ендпойнт);

– прийнявши запит від клієнта на обробку, контролер викликає відповідний метод із BLL-сервісу;

– після попередньої валідації Request-моделі, сервіс викликає необхідний метод з інтерфейсу, згенерованого CSnakes для виконання Python-скриптів в рантаймі;

– отримавши результати та повернувши їх в коректній Response-моделі для даної задачі, на стороні клієнта створюється нова модель із записом про обрахунок, що буде передана на користувацьке API;

– сформувавши модель та передавши її як параметр в метод відповідного HTTP-сервісу, спочатку ми перевіряємо, чи користувач зараз автентифікований в системі, чи ні. Якщо це дійсно так, то перед оновленням самого графіку та відображення результатів, буде надіслано запит до User API на відповідний контролер;

– після додавання нового запису в таблицю з історією, також має бути додатково перевірено поточну кількість вже наявних рядків для цього користувача. Якщо ж кількість записів більше або дорівнює 100, то з бази має бути видалено найдавніший запис. Після успішного видалення користувачу буде продемонстровано результати виконання чисельного методу. Дана логіка буде застосовуватися до всіх видів обрахунків, окрім запитів на порівняння методів (в такому випадку, історія в будь-якому разі не зберігатиметься).

Тепер розглянемо архітектуру користувацького сервісу на рисунку В.5 з додатку В.

Як і у випадку із Calculation API, тут також будуть наявні контролери у відповідності до необхідного функціоналу:

– контролер, що відповідатиме за створення нових користувачів, їхню автентифікацію, та відновлення забутого пароля;

– окремий контролер для керування користувацькою інформацією (включно з видаленням зареєстрованого облікового запису);

– контролер для отримання даних по останніх проведених обрахунках та для створення нового запису в базі;

– контролер для керування збереженими введеними параметрами;

– контролер для керування останніми завантаженими файлами.

Окрім бізнес-сервісів під кожен наведений вище контролер, також треба буде створити окремий сервіс, що дозволить надсилати повідомлення користувачам за відповідною e-mail адресою. Дане завдання вимагає попереднє налаштування окремої електронної пошти для проєкту, від імені якого листи й будуть відправлятися.

На відміну від системи обчислень, User API має так звані репозиторії, що

в багатошаровій архітектури називаються data access layer (DAL). Даний шар дозволяє відокремити логіку з управлінням контексту бази даних (DB context) подалі від бізнес-логіки. Відповідно цей контекст дозволяє визначати структуру бази на рівні C#-коду, без необхідності в створенні окремого проєкту саме на структуру бази (SQL projects).

Для реалізації самого сервісу, було вирішено використати таку архітектуру, як Clean Architecture, яка де-факто є однією із стандартних моделей про побудові користувацької бекенд-частини. За цим принципом загальна бізнес-логіка має бути ізольована від зовнішніх частин, а всі залежності мають йти з середини назовні. Завдяки цьому паттерну можна максимально абстрагуватися від таких потенційних залежностей, як СКБД, фреймворки ітд., що в свою чергу дозволяє швидко писати юніт-тести на перевірку валідності сервісів.

2.3 Створення моделі бази даних з користувачами

Основні вимоги проєктування включають в себе зберігання таких користувацьких даних:

– автентифікаційні дані, як логін, пароль, електронна пошта;

– записи про останні обрахунки, окрім тих, що відбувалися для порівняння чисельних методів;

– останні завантаження результатів обчислень з можливістю повторного завантаження;

– збережені початкові параметри для обрахунків.

Розглянемо основні таблиці та зв’язки, які потрібні для належного функціонування системи. Для цього можна скористатися Entity Relationship діаграмою на рисунку 2.1.

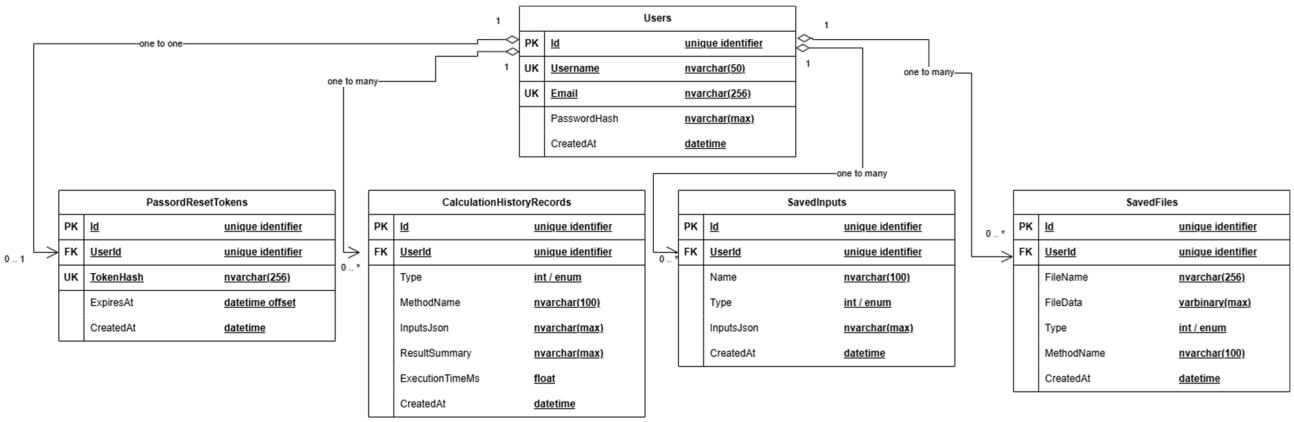


Рисунок 2.1 – ER-діаграми для опису будови користувацької бази даних

Основною сутністю тут є користувач, який представлений у формі таблички Users. Кожна інша таблиця має зовнішній ключ саме головний ключ даної таблиці – ID, що виступає головним ідентифікатором користувача. Поле CreatedAt, що представляє дату створення користувача, також автоматично буде проставлятися за допомогою SQL-функції «GETUTCDATE()». Поля з часовими значеннями записуються з врахуванням саме UTC-часу, що вважається стандартом для уніфікації часу в межах різних часових поясів. Поля Username та Email мають бути унікальними в межах таблиці, тому для них також буде створено унікальний ключ. Головний ключ також передбачає унікальність, тому полю Id не треба було явно зазначати створення унікального ключа. Для хорошої індексації, ключі мають обмеження на довжину: логін не може містити більше 50-ти символів, а електронна пошта – більше 256-ти символів.

Зазвичай, при створенні таблиць користувачів створюють окрему колонку для зберігання солі, що враховується при хешуванні пароля. Хоча ця сіль потрібна для того, аби забезпечити унікальність хешу в межах можливих однакових паролів користувачів, проте тут це не є необхідністю, оскільки бібліотека BCrypt дозволяє записувати сіль прямо в хеш, та витягувати її, при перевірці хеш-значення.

В межах користувацької поведінки серед вимог є можливість відновлення забутого паролю. Даний функціонал вимагатиме зберігання окремих токенів, що відповідатимуть саме за можливість відновлення паролю після запиту в межах 30 хвилин. На діаграмі ця таблиця для зберігання представлена як

PassordResetTokens, що містить наступні поля:

– Id – головний ідентифікатор запису;

– UserId – ідентифікатор користувача;

– TokenHash. Захешований токен, із застосуванням SHA-256 алгоритму, що відправлятиметься клієнтською частиною разом із введеним новим паролем.

Поля CreatedAt та ExpiresAt проставлятимуться автоматично або на стороні SQL Server (за допомогою «GETUTCDATE()»), або на рівні C#-коду (як у випадку з полем ExpiresAt). Якщо користувач перейшов на сторінку із невалідним токеном, то клієнтській частині буде повернуто 400-ий HTTP статус-код. Так само буде відбуватися, якщо токен дійсно в базі наявний, але час його використання завершився (30 хвилин).

Розглянемо наступні таблиці: CalculationHistoryRecords, SavedInputs та SavedFiles. Спільне між ними те, що всі вони мають ідентифікатори, типу UNIQUEIDENTIFIER, та зовнішній ключ на полі UserId, що містить посилання на табличку Users. Якщо в межах таблиці з токенами для відновлення пароля дозволялося мати тільки один запис для одного користувача (зв’язок «один до одного»), то останні згадані допускають створення багатьох записів для одного користувача (зв’язок «один до багатьох»). Поле CreatedAt також присутнє скрізь і задається воно автоматично при додаванні нових записів до таблиць.

CalculationHistoryRecords представляє таку сутність, як запис з історії обрахунків. Даний запис включатиме в себе тип (відповідає типу математичної задачі, назву методу (має обмеження до 100 символів в довжину), значення JSON з введеними параметрами та результатом обрахунків, час виконання алгоритму в мілісекундах та, звичайно, дату створення даного запису. Однією з вимог було дозволити максимально тільки 100 записів на одного користувача в межах історії обчислень. Оскільки стандартні інструменти SQL Server цього не дозволяють, було вирішено винести дану валідацію на рівень майбутнього бізнес-сервісу.

Таблиця, із збереженими параметрами для обчислень, містить менше полів:

– Name. Поле, що репрезентує назву, яка задаватиметься користувачем під час створення нового запису. Має обмеження до 100 символів;

– Type. Тип математичного завдання;

– InputsJson. JSON-значення, що є словником зі значень параметрів. Під час застосування певного збереженого значення, саме звідти всі необхідні для заповнення поля й беруться.

На відміну від CalculationHistoryRecords та SavedFiles, до даної таблиці не виставлено вимог по максимальній можливій кількості зберігання рядків.

Остання таблиця, яка має бути розглянутою – SavedFiles. Саме тут зберігатимуться останні завантажені користувачем результати виконання завдань, заповнюючи такі поля, як FileName (назва файлу, що генерується автоматично системою), FileData (масив байтів з усією інформацією) та MethodName (назва методу алгоритму, що використовувалася для обчислення). Зауважимо, що колонки FileName та MethodName мають також обмеження в довжинах: 256 символів для назви файлу та 100 символів для назви методу.

Для реалізації зберігання користувацьких даних було вирішено застосувати MS SQL Server, як СКБД, оскільки вона легко інтегрується з .NET-технологіями, зокрема з використанням такого пакета, як EntityFrameworkCore.SqlServer, що є офіційним продуктом Microsoft. Перевагами MS SQL Server також є:

– забезпечення реляційної цілісності (ACID-транзакції);

– шифрування з’єднань із застосуванням TLS (криптографічний протокол);

– наявність зрозумілої документації;

– кросплатформність.

Таким чином спроєктовані сутності з усіма необхідними зв’язками та ключами дозволять забезпечити цілісність даних та зменшити кількість потенційних помилок пов’язаних із зберіганням користувацької інформації. Додані ключі дозволять забезпечити кращу швидкодію при пошуку записів, оскільки на базі них Entity Framework Core при публікації бази автоматично створить індекси. Вище спроєктовані таблиці повністю покриватимуть необхідний функціонал для досягнення вимог, зазначених в попередньому підрозділі.

РОЗДІЛ 3
РОЗРОБКА ІНФРАСТРУКТУРИ ДЛЯ ВИКОНАННЯ ОБРАХУНКІВ ЧИСЕЛЬНИМИ МЕТОДАМИ

3.1 Практична реалізація об’єкта проєктування. Побудова блок-схем модулів програми

Практична реалізація системи розпочалася із початкової розбудови серверної інфраструктури: API для обрахунків та користувацьких даних. На початку, розглянемо загальну структуру проєкту (рис. 3.1).

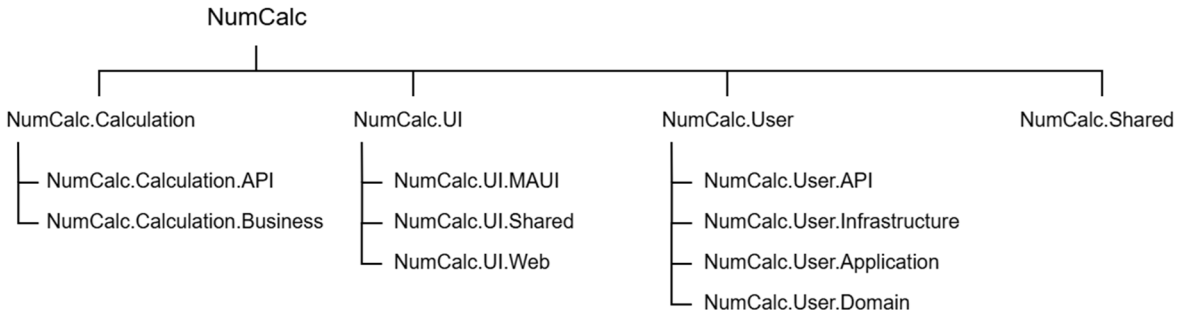


Рисунок 3.1 – Структура проєкту на рівні кодової бази

Система складається з 4 основних підсистем, які в сукупності мають 10 кодових проєктів:

– NumCalc.Calculation. Є втіленням API для проведення обрахунків;

– NumCalc.UI. Спільна solution-папка, що містить такі проєкти, як NumCalc.UI.Web (вебсервіс), NumCalc.UI.MAUI (кросплатформний застосунок) та NumCalc.UI.Shared (Razor Class Library, яка буде спільною кодовою базою із компонентів та сторінок для вебсервісу та застосунків);

– NumCalc.User. Уособлення API для керування користувацькими даними;

– NumCalc.Shared. Містить спільні DTO-моделі (включно з Request- та Response-моделями) для таких проєктів, як NumCalc.Calculation та NumCalc.UI.Shared. Дана класова бібліотека дозволить уникнути зайвого дублювання коду і зменшити ймовірність можливої помилки при зміні моделей.

Оскільки на початку було вирішено ознайомитися з серверною

інфраструктурою, розглянемо систему обрахунків (NumCalc.Calculation). Даний проєкт побудований за принципом багатошарової архітектури (N-layered Architecture) та містить таких два проєкти: NumCalc.Calculation.API та NumCalc.Calculation.Business. Перший проєкт виступає як презентаційний шар, а другий – шар бізнес-логіки. В NumCalc.Calculation.API, який залежний від BLL-сервісу, лежить основний файл для запуску проєкту – «Program.cs». Даний файл має налаштування, необхідні для:

– логування подій та помилок за допомогою Serilog;

– підключення контролерів та генерації документації для Swagger на основі summary-коментарів;

– завантаження пакетів, що необхідні для Python-скриптів, та підключення самого CSnakes. Останньому ви вкажемо, де лежать всі скрипти та створене окреме віртуальне Python-середовище;

– реєстрації сервісів з NumCalc.Calculation.Business в DI-контейнері;

– під’єднання глобального обробника помилок.

Всі події та помилки, що виникатимуть в процесі роботи сервісу, будуть записуватися у відповідну папку «Logs» в обмеженні до 14-ти можливих файлів. Загалом розраховано, що один файл буде містити дані за один день. Нижче, в лістингу 3.1, наведено конфігурацію CSnakes в «Program.cs».

Лістинг 3.1 – Налаштування CSnakes в «Program.cs»

```
var binPath = AppDomain.CurrentDomain.BaseDirectory;
var projectRoot = Path.GetFullPath(Path.Combine(binPath,
"../../../../"));
var venvPath =  builder.Configuration["Python:VenvPath"]
                ?? Path.Combine(projectRoot, ".venv");
var scriptsPath = Path.Combine(binPath, "Scripts");
builder.Services
    .WithPython().WithHome(scriptsPath)
    .WithVirtualEnvironment(venvPath).FromRedistributable()
    .WithPipInstaller(Path.Combine(scriptsPath,
"requirements.txt"));
```

кінець лістингу 3.1

Тут вказується шлях, за яким має бути віднайдено наші скрипти для

генерації C#-інтерфейсів, та шлях до файлу «requirements.txt», де вказані пакети. Згенеровані інтерфейси будуть використовуватися в сервісах. Пакети підключатимуться з використанням pip – системи керування пакетами для Python-середовища (схоже на npm в Node.js та NuGet в C#).

Для імплементації усіх методів в систему треба підключити наступні пакети:

– SymPy. Дозволяє перетворювати звичайні математичні вирази, записані в текстовому значенні, в символьні об’єкти. LaTeX-значення, що будуть репрезентувати формули в згенерованих покрокових рішеннях, також будуть утворені за допомогою цієї бібліотеки;

– NumPy. Має функції необхідні для розбиття заданого проміжку на менші шматки (для інтегрування), проведення матричних операцій (для лінійних систем рівнянь) тощо;

– SciPy. Містить алгоритми, необхідні для кубічного сплайну в межах інтерполяції функцій;

– mpmath. Є транзитивною залежністю SymPy, яка хоч і ніде не використовується, але була необхідна для імпортування;

– latex2sympy2. В контексті системи дозволяє перетворювати математичні виразі, що записані в LaTeX-форматі, в SymPy-вираз.

Всі сервіси з NumCalc.Calculation.Business реєструються як Scoped, що є дефолтом для API-сервісів (на кожен запит створюється новий екземпляр класу, який потім знищується так званим «збирачем сміття»). Глобальний обробник помилок дозволить відловлювати будь-які винятки, які виникатимуть в межах контролерів, та визначати їхній тип, повертаючи в результаті або 400-ий (на випадок виникнення проблеми внаслідок невалідних даних зі сторони клієнта), або 500-ий HTTP статус-код (якщо помилка виникла по причині сервера). В лістингу 3.2 наведено кодову реалізацію даного global exception handler з реалізованим методом TryHandleAsync.

Лістинг 3.2 – Глобальний обробник винятків на Calculation API

```
public class
GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger) :
IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(HttpContext
httpContext, Exception exception, CancellationToken
cancellationToken)
    {
        var problemDetails = new ProblemDetails() { Instance =
httpContext.Request.Path };
        if (exception is CustomException calculationException)
        {
            logger.LogWarning("Validation error: {Message}. Code:
{Code}", exception.Message, calculationException.ErrorCode);
            FillClientProblemDetails(problemDetails,
calculationException);
        }
        else
        {
            logger.LogError(exception, "Unhandled exception
occurred: {Message}", exception.Message);
            FillServerProblemDetails(problemDetails, exception);
        }
        httpContext.Response.StatusCode =
problemDetails?.Status ??
StatusCodes.Status500InternalServerError;
        await
httpContext.Response.WriteAsJsonAsync(problemDetails,
cancellationToken);
        return true;
    }

    #region Private methods ... #endregion
}
```

кінець лістингу 3.2

Тут CustomException – дочірній клас Exception (namespace – System), що містить додаткову властивість – ErrorCode. Дана властивість представляє уніфікований тип певної помилки на основі створеного перелічення (enum) – NumCalcErrorCode. Таким чином можна повертати наперед визначені можливі помилки, спростивши їхню обробку на клієнтській стороні (зокрема в локалізації помилок для користувача). Структуру даного перелічення наведено в лістингу 3.3, яка може з часом змінюватися в разі розширення функціоналу системи.

Лістинг 3.3 – Визначення перелічення NumCalcErrorCode

```
[JsonConverter(typeof(JsonStringEnumConverter))]
public enum NumCalcErrorCode
{
    [JsonStringEnumMemberName("RANGE_INVALID")]
    RangeInvalid,
    [JsonStringEnumMemberName("SYNTAX_ERROR")]
    SyntaxError,
    [JsonStringEnumMemberName("ZERO_DIVISION")]
    ZeroDivision,
    [JsonStringEnumMemberName("TIMEOUT")]
    Timeout,
    [JsonStringEnumMemberName("UNKNOWN_ERROR")]
    UnknownError,
    [JsonStringEnumMemberName("INVALID_JSON")]
    InvalidJson,
    [JsonStringEnumMemberName("EMPTY_RESPONSE")]
    EmptyResponse,
    [JsonStringEnumMemberName("NOT_IMPLEMENTED")]
    NotImplemented,
    [JsonStringEnumMemberName("EMPTY_DATA")]
    EmptyData,
    [JsonStringEnumMemberName("INVALID_DATA")]
    InvalidData
}
```

кінець лістингу 3.3

Варто зазначити, що проєкт NumCalc.Calculation.API також містить «Dockerfile» – скрипт, який необхідний для збірки Docker-image даного сервісу. Створений image може використовуватися як під час локального тестування, так і для розгортання проєкту на іншому віддаленому сервері.

Розглянемо наступний проєкт – NumCalc.Calculation.Business, який уособлює BLL (Business Logic Layer) в архітектурі системи. Тут наявні наступні важливі папки:

– «Entities». Моделі даних, що вертаються у форматі JSON після обрахунків. Кожна модель згруповується у відповідну свою DTO-модель в межах задачі, та вертаються клієнтській стороні.

– «Exceptions». Містить вищезгаданий клас CustomException, який якраз також використовується в сервісах бізнес-логіки (зокрема при валідації моделей).

– «HostedServices». Містить сервіси, які мають виконатися при запуску API.

– «Scripts». Тут знаходяться всі написані Python-скрипти, що необхідні для

виконання обчислень різними методами.

– «Services». Сервіси бізнес-логіки, кожен з яких відповідає за пошук розв'язку однієї конкретної математичної задачі. Сюди ж також відноситься і сервіс для розпізнавання математичних виразів по зображенню.

– «Utils». Статичні допоміжні класи.

В «Entities» під кожну математичну задачу виділено свою підпапку для зберігання моделей. Серед важливих моделей можна виділити наступні дві: PythonResponse та PythonFailureData. Перша модель репрезентує результат, що вертається в усіх Python-скриптах (ліст. 3.4).

Лістинг 3.4 – Модель результату виконання методу

```
public class PythonResponse<T>
{
    public T? Success { get; set; }
    public PythonFailureData? Failure { get; set; }
}
```

кінець лістингу 3.4

T в PythonResponse – тип моделі, яка вертається при успішному виконанні метода. Оскільки тут немає якогось єдиного варіанту відповіді (наприклад, при розв'язку системи рівнянь вертається скалярне значення, а при інтерполяції – поліном), то вирішено було скористатися наступним паттерном – «Result Pattern». Даний шаблон проєктування пропонує замість повторного викидання винятків повертати спеціальний об'єкт, що є менш затратною операцією. У випадку виникнення помилки, в методах, на стороні Python, catch-блоки вертатимуть об'єкт, що має заповнену властивість Failure. Клас PythonFailureData має: Code та Message. Code – властивість, що відповідає JSON-назві значення з NumCalcErrorCode, а Message – опис помилки. Для перетворення та отримання результату виконання є написаний метод UnwrapOrThrow, що є частиною такого допоміжного класу, як PythonResponseUtils. Структуру класу та його методів описано в лістингу Г.1. Вказані в ньому JsonOptions дозволяють коректно спарсити Code у відповідне значення NumCalcErrorCode.

IHostedService – інтерфейс в .NET-платформі, що призначений для створення фонових сервісів, які мають виконатися на початку запуску застосунків або протягом всього його життєвого циклу (так звані BackgroundService). При розбудові системи було зауважено, що першопочатковий запит на обчислення дуже довго виконується: в середньому до 30 секунд. Це відбувається за рахунок того, що CSnakes Runtime створює необхідне Python-середовище тільки тоді, коли хтось «звернувся» до нього, оскільки проходить процес «лінивого завантаження» (lazy loading). Зважаючи на його можливий негативний вплив на UX вебплатформи та нативних MAUI-додатків, було вирішено одразу зробити звернення до CSnakes Runtime перед стартом API (ліст. 3.5).

Лістинг 3.5 – Опис класу PythonWarmingUpService

```
public sealed class PythonWarmingUpService(IPythonEnvironment env,
ILogger<PythonWarmingUpService> logger) : IHostedService
{
    public async Task StartAsync(CancellationToken
cancellationToken)
    {
        logger.LogInformation("Python warmup starting");
        await Task.Run(() =>
        {
            var warmingUpService = env.WarmingUp();
            warmingUpService.Run();
        }, cancellationToken);
        logger.LogInformation("Python warmup completed");
    }

    public Task StopAsync(CancellationToken cancellationToken) =>
Task.CompletedTask;
}
```

кінець лістингу 3.5

В самому методі зі скрипта («Scripts/warming_up.py») відбувається імпорт всіх необхідних для роботи пакетів та повертається ціле число (одиниця), як знак успішного підвантаження пакетів та виконання функції. Допоки StartAsync всіх сервісів, що є підключені як Hosted Service в «Program.cs», не будуть виконані, до Calculation API неможливо звернутися: специфіка роботи Kestrel.

На останок розглянемо загальну схему роботи кожного публічного методу в сервісах бізнес-логіки, що стосуються саме обрахунків (рис. 3.2).

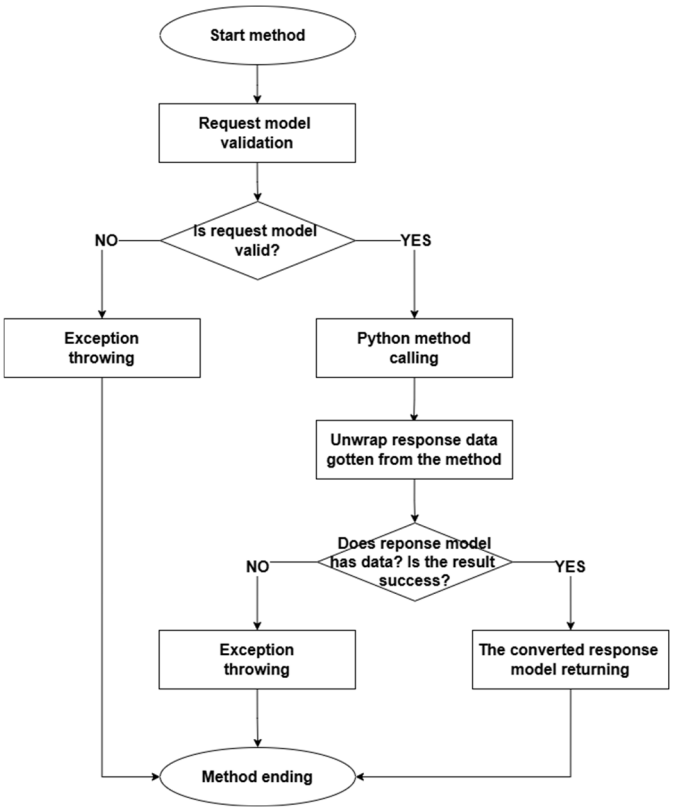


Рисунок 3.2 – Блок-схема виконання методів в обчислювальних BLL-сервісах

Кожен метод на початку має провалідувати Request-модель, сформовану клієнтською частиною та отриману з тіла HTTP-запиту. У випадку некоректності даних в методі викидатиметься виняток типу CustomException (з передачею значення самої помилки та її опису). Пізніше підключений global exception handler перехопить цю помилку та поверне клієнтській частині відповідь з 400-им HTTP статус-кодом. Якщо ж дані прийшли валідні, то буде викликано необхідну функцію зі створеного Python-інтерфейсу, з попередньою передачею параметрів. Після отримання результатів, у випадку успішності виконання алгоритму, контролеру повертатиметься сформована DTO-модель на основі даних самого результату. Якщо ж щось відбулося не те під час виконання скрипта, і було повернуто модель PythonResponse із заповненим Failure, то буде сформовано та викинуто CustomException. Приклад одного із реалізованих обчислювальних методів можна побачити в лістингу 3.6.

Лістинг 3.6 – Реалізований метод бісекції в сервісі для пошуку нуля функції

```
public RootFindingResponse CalculateDichotomy(RootFindingRequest
request)
{
        ValidateRequest(request);

        logger.LogInformation("Dichotomy: f={Expression},
[{Start}, {End}], err={Error}",
            request.FunctionExpression, request.StartRange,
request.EndRange, request.Error);

        var rootSolver = env.RootFinding();

        var jsonEnvelope = rootSolver.SolveDichotomy(
            request.FunctionExpression,
            request.StartRange,
            request.EndRange,
            request.Error
        );

        var rootData =
jsonEnvelope.UnwrapOrThrow<RootFindingData>();
        logger.LogInformation("Dichotomy completed: root={Root},
iterations={Iterations}",
            rootData?.Root, rootData?.Iterations);

        return new RootFindingResponse
        {
            Root = rootData?.Root,
            Iterations = rootData?.Iterations ?? 0,
            ChartData = rootData?.ChartPoints,
            SolutionSteps = rootData?.SolutionSteps
        };
}
```

кінець лістингу 3.6

Варто оглянути й NumCalc.Shared – класову бібліотеку, що містить всі DTO-моделі, пов’язані з побудованим обчислювальним API. Окрім розміщених тут Request- та Response-моделей, тут також є описані класи, які становлять частину результатів майже всіх обчислень: Point та SolutionStep.

Перший клас представляє точку, яка може знаходитися як у 2D-, так і у 3D-просторах (як у прикладі з пошуком екстремуму функції, або системами рівнянь). SolutionStep – модель, що дозволить відобразити певний етап виконання алгоритму або ітерації. Вона має наступні властивості:

– StepIndex – порядковий номер в списку;

– Description – короткий опис етапу;

– LatexFormula – важлива обчислена формула, яку варто було б відобразити;

– Value – обраховане певне значення, як результат виконання даного етапу.

Більшість моделей-відповідей містять таке поле, як ExecutionTimeMs – час виконання алгоритму. Для його фіксації використовується клас Stopwatch з System.Diagnostics, екземпляр якого запускається в момент старту виклику методу та зупиняється після отримання результату. При поверненні DTO-моделі, відповідний показник фіксується (в мілісекундах). Відносно моделей порівнянь можна зауважити наступне. Усі з них зазвичай повертатимуть те ж саме: масив з результатами, аналогічних до результату виконання одинарного алгоритму, та найпродуктивніший метод серед тих, що застосувалися при порівнянні. Якщо ж деякі Response-моделі містять габаритні дані, по типу ChartData або SolutionSteps, то вони будуть відсутні у відповіді при порівнянні методів. Це було б зайвою інформацією, яка до того ж могла бути великою в розмірі.

Як вже раніше згадувалося, всі Python-скрипти знаходяться в папці «Scripts». Кожен метод розміщений в своєму окремому файлі, щоб була забезпечена краща читабельність для інших розробників. Важливою тут також є наступна папка – «shared», де знаходяться моделі для:

– формування результатів вирішення задач та безпосередньо PythonResponse (шлях «Scripts/shared/structures.py»);

– перетворення виразу, записаного в LaTeX-форматі, в SymPy-вираз (шлях «Scripts/shared/parsing.py»);

– загальні спільні функції (шлях «Scripts/shared/functions.py»);

– функції, що мають використовуватися при побудові графіків при розв’язку систем рівнянь 2-ої та 3-ої розмірності (шлях «Scripts/shared/equation_system_charts.py»).

Перед написанням обчислювальних методів важливо було ознайомитися із чисельними методами, які плануються до реалізації. Спочатку розглянемо задачу з пошуку кореня рівняння, та діаграми на рисунку 3.3, що описують основні моделі, що фігурують в даній площині обрахунків.

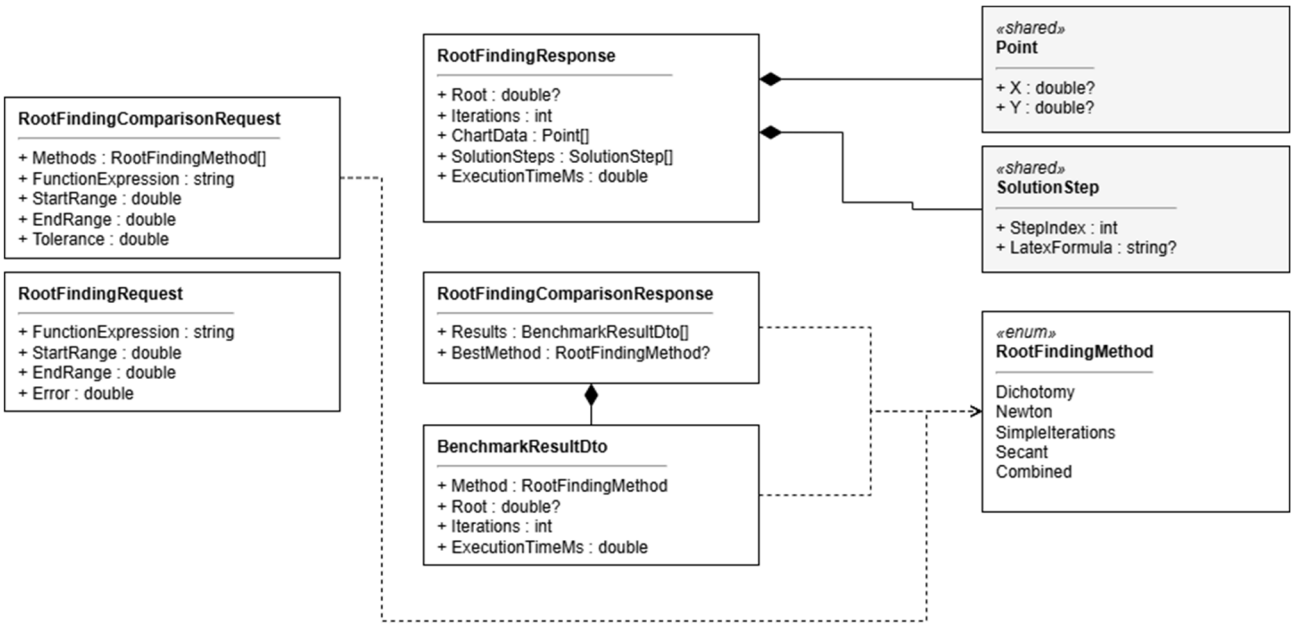


Рисунок 3.3 – Моделі, що стосуються задачі з пошуку нуля функції

Основними моделями, що стосуються саме запитів, є RootFindingRequest та RootFindingComparisonRequest. Якщо перша відноситься до пошуку нуля функції саме із застосуванням одного конкретного чисельного методу, то друга відноситься до порівняння. Спільними властивостями тут є:

– FunctionExpression – текстове значення функції, яке може представлятися в LaTeX-форматі також;

– StartRange – початкова точка відрізка, на якому шукатиметься нуль функції;

– EndRange – кінцева точка відрізка;

– Tolerance – допустима похибка результату, досягнення якої завершує виконання алгоритму.

Для методу Ньютона, де потрібно замість меж вказувати наближене значення, відсутня окрема властивість. В такому разі за наближення обирається вказаний початок досліджуваного відрізку. RootFindingComparisonRequest має таке поле, як Methods, що є масивом з методів, які будуть порівнюватися.

Зі сторони сервера при обрахунках повертатиметься RootFindingResponse, що містить такі властивості:

– Root – корінь рівняння;

– Iterations – кількість здійснених ітерацій;

– ChartData – масив точок для можливої побудови графіка;

– SolutionSteps;

– ExecutionTimeMs.

Якщо відбувалося порівняння моделей, то вже буде повертатися інша модель – RootFindingComparisonResponse. Для методу Ньютона похідна визначається з використанням бібліотеки SymPy, після чого вона компілюється для швидких подальших розрахунків.

Другим типом задач є пошук розв’язку систем лінійних та нелінійних рівнянь – віднаходження масиву значень, які при підстановці в невідомі змінні рівнянь задовольнятимуть систему. Якщо у випадку пошуку нуля функції можна було б скористатися графічним методом, то тут ситуація складніша: особливо у випадку багатовимірних просторів. В таких умовах чисельні методи мають велику перевагу – зокрема при розв'язку систем нелінійних рівнянь. Під час розробки системи було реалізовано такі методи:

– метод Крамера для розв’язку СЛАР. Цілком застосовується у тому випадку, якщо кількість рівнянь збігається з кількістю невідомих;

– метод Гауса, що, на відміну від методу Крамера, дозволяє розв’язувати систему лінійних рівнянь будь-якої порядку;

– метод простих ітерацій, що ґрунтується на послідовному наближенні до результату (як і у випадку з пошуком кореня рівнянь);

– метод Зейделя – модифікація методу простих ітерацій.

Хоча й перші два методи є аналітичними, проте було вирішено винести їх також для обрахунків, зважаючи на їхню популярність у використанні та часткову ітераційну природу. Розглянемо реалізовані методи для пошуку розв’язку систем у випадку лінійних рівнянь.

Якщо в методі Крамера визначається, що головний визначник дорівнює нулю, клієнтській частині буде повернуто 400-ий HTTP статус-код одразу, запобігаючи виникненню помилки при діленні на нуль. В реалізованому Python-методі для Гауса прямий хід відбувається за схемою вибору головного елемента

по стовпцю, переставляючи найбільші за модулем елементи наверх перед обнуленням стовпця. Техніку часткового півотингу описано у формулі 3.1.

$$R_i \leftrightarrow R_j. \tag{3.1}$$

Використання цього методу забезпечує зменшення можливих похибок та підвищує стабільність алгоритму. Повний півотинг, що включає також перестановку стовпців, не використовується, оскільки переміщення рядків також достатньо для більшості задач. Також частковий варіант є більш обчислювально ефективним, адже не виконує зайвих операцій.

Метод простих ітерацій за своєю природою майже аналогічний до однойменного алгоритму для пошуку коренів рівнянь. Вхідна система має бути зведена до вигляду, в якому кожна невідома виражена через відповідні функції. Якщо у випадку алгоритму пошуку нуля функції, для запобігання розбіжності, в Python-скрипті попередньо віднаходять ітераційний параметр, то в ситуації з системами рівнянь будують матрицю Якобі. Дана матриця заповнюється похідними від векторних функцій, що може бути достатньо дорогою операцією в плані ресурсів сервера. Тому в даній реалізації простих методів ця матриця не використовується: користувачеві вертатиметься відповідна помилка за умови виявлення розбіжності.

На рисунку 3.4 відображено спільні моделі, що використовуються в межах розв’язку систем лінійних рівнянь.

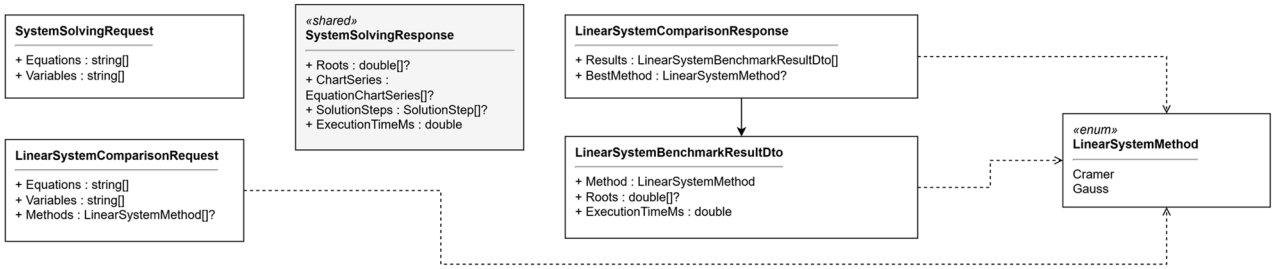


Рисунок 3.4 – DTO-моделі, пов’язані із розв’язуванням систем рівнянь

SystemSolvingRequest дозволяє передати введені користувацькі значення:

масив з рівняннями та самі змінні. Тут Equations – введені лінійні рівняння, а Variables – масив із змінними системи. LinearSystemComparisonRequest – обгортка навколо даних, що передаються в запиті на порівняння методів. Окрім попередньо згаданих ідентичних властивостей, як у SystemSolvingRequest, дана модель також передає список методів, які потрібно порівняти (Methods). Якщо було успішно вирішено задачі, незалежно від типу рівнянь, то повертатиметься загальна модель SystemSolvingResponse, де Roots – список коренів, ChartSeries – точки для побудови 2D- та 3D-графіків. Відповідь порівняння методів – LinearSystemComparisonResponse – містить список з результатів (LinearSystemBenchmarkResultDto) та найкраще відпрацьований метод.

Моделі, пов’язані із системами нелінійних рівнянь, є майже ідентичними (рис. 3.5).

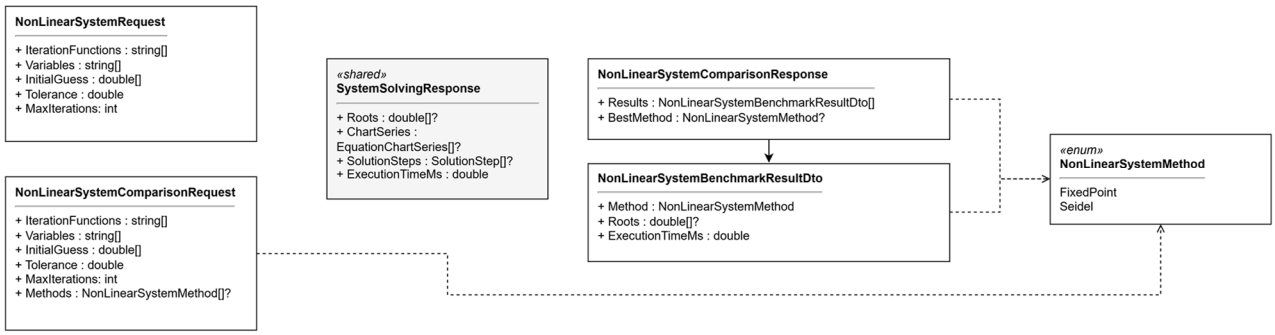


Рисунок 3.5 – DTO-моделі, пов’язані із системами нелінійних рівнянь

Однієї відмінністю є різниця у властивостях Request-моделей. Тут моделі замість рівнянь та змінних приймають наступні значення:

– IterationFunctions – ітераційні функції визначені користувачем;

– Variables – змінні, що використовуються в системі;

– InitialGuess – початкове наближення;

– Tolerance – допустима похибка;

– MaxIterations – допустима максимальна кількість ітерацій.

Структура моделей запиту та відповіді відносно порівняння методів є схематично ідентичною.

Інтерполяція – процес віднаходження невідомих проміжних значень на

основі певного дискретного набору. Функція, яка будується при інтерполяції, має точно проходити через вказані координати. Чисельні методи дозволяють використовувати простіші функції (наприклад, базисні поліноми) замість аналітичних складних, що зменшує обчислювальні витрати та дозволяє експериментувати з такими даними, які неможливо або складно описати конкретною формулою.

Побудована серверна частина надає можливість скористатися наступними методами:

– Лагранжа, що ґрунтується на створенні інтерполяційного полінома через базисні поліноми;

– Ньютона, що базується на побудові полінома з використанням розділених різниць;

– кубічних сплайнів, що будує сукупність кубічних многочленів на кожному інтервалі між вузлами.

Варто ознайомитися з моделями на рисунку 3.6, які фігурують в домені інтерполяції.

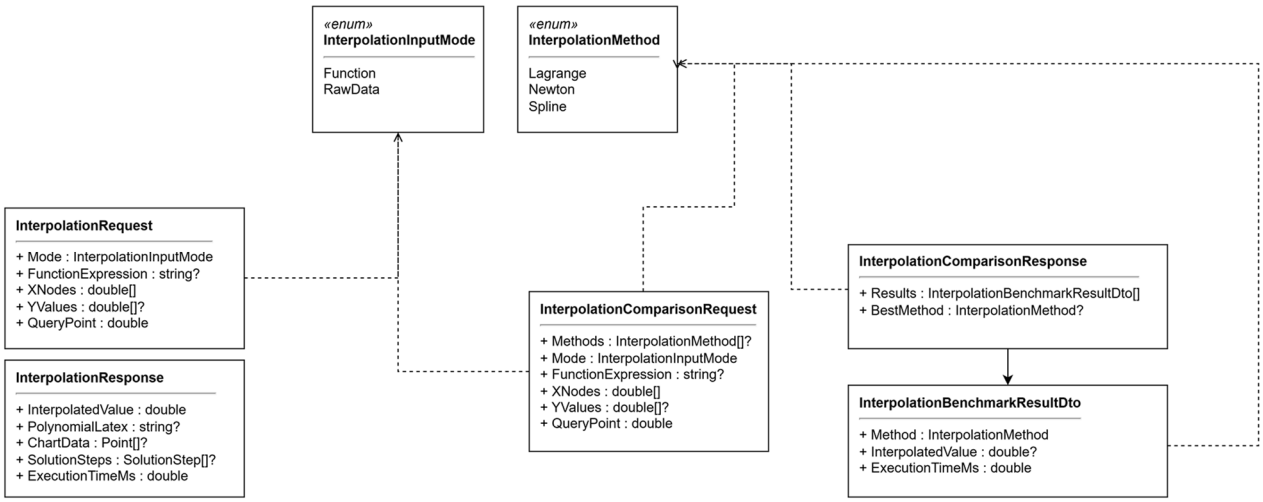


Рисунок 3.6 – DTO-моделі, пов’язані з інтерполяцією

Найцікавішою властивістю в InterpolationRequest є Mode. Вона вказує на те, в якому вигляді серверу передалися дані: чи суто дискретний набір координат (□, □) (XNodes та YValues), чи список □-вузлів (XNodes), проте з додатковим

зазначенням текстового опису функції (FunctionExpression). Хоча й в межах системи, мабуть, достатньо було передавати тільки дискретний набір, проте таке розширення, з передачею формули функції, робить функціонал набагато гнучкішим, і знижує ймовірність введення некоректних даних користувачем. На відміну від попередніх Response-моделей, InterpolationResponse також містить окреме поле для відображення побудованого інтерполяційного поліному в межах виконання обраного чисельного методу.

Чисельне диференціювання – це обрахунок похідної неперервної функції, що ґрунтується на основі заданого дискретного набору. Воно є незамінним у випадку, коли функцію тяжко описати певною формулою, або її складно обчислити саме аналітично. В межах Calculation API реалізовано декілька можливих для вибору методів:

– скінченних різниць (прямі, центральні, зворотні). Основа – різниці значень функції в сусідніх вузлах;

– метод Лагранжа. Після побудови інтерполяційного поліному Лагранжа через вказані точки, метод проводить символьне диференціювання навколо нього, повертаючи похідну, як функцію від □.

Недоліком методу скінченних різниць є неможливість повноцінно працювати з дискретним набором даних: похідні в точках можна було б знайти тільки у вказаних в таблиці вузлах. Тому було вирішено, для методу скінченних різниць, дозволити користувачеві передавати тільки вираз функції, без можливості вказання дискретного набору. Написаний Python-метод, що відповідає за обрахунки похідних з використанням методу Лагранжа, використовує можливості такого пакету, як SymPy, для отримання аналітичного рівняння похідної (викликана функція – «sympy.diff(poly_expanded, x)»). Отримане рівняння також передаватиметься як частина результату клієнту.

На діаграмі нижче (рис. 3.7) відображено моделі, що використовуються для вирішення задач з диференціювання функції.

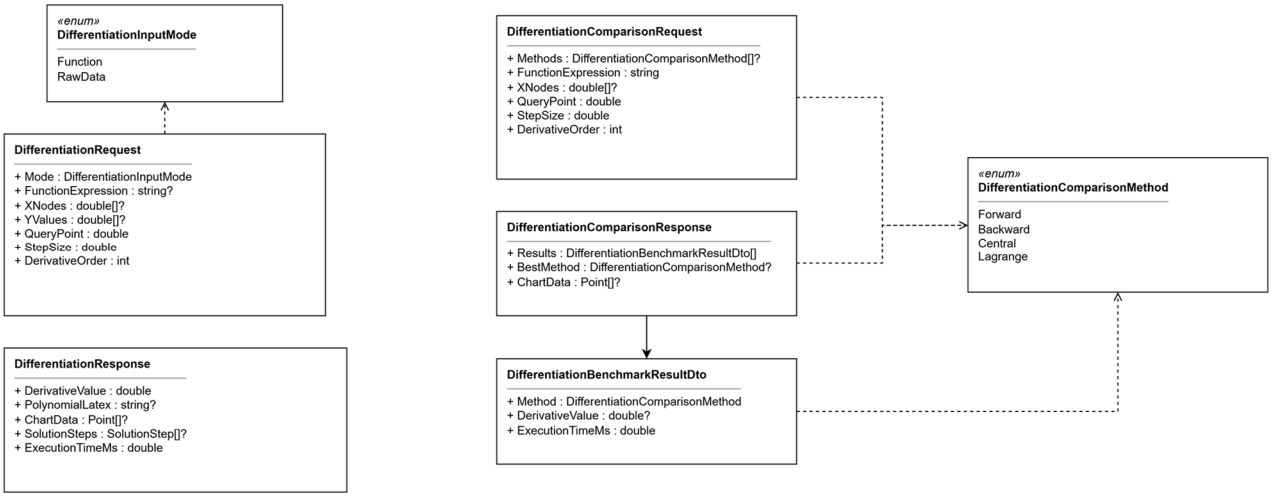


Рисунок 3.7 – DTO-моделі, пов’язані з диференціюванням

Порядок похідної передається у властивості DerivativeOrder, що є в моделі для звичайних одинарних обрахунків (DifferentiationRequest), і для порівняння (DifferentiationComparisonRequest). Для вибору конкретного типу алгоритму скінченних різниць, клієнт має передати в запит обраний варіант (ліст. 3.7).

Лістинг 3.7 – Реалізований контролер для можливості диференціювання

```
[ApiController]
[Route("api/[controller]")]
[Produces("application/json")]
public class DifferentiationController(IDifferentiationService
differentiationService) : ControllerBase
{
    [HttpPost("finite-diff")]
    [ProducesResponseType(typeof(DifferentiationResponse),
StatusCodes.Status200OK)]
    [ProducesResponseType(typeof(ProblemDetails),
StatusCodes.Status400BadRequest)]
    public IActionResult SolveFiniteDiff([FromBody]
DifferentiationRequest request, [FromQuery] FiniteDiffVariant
variant = FiniteDiffVariant.Central)
    {
        var response =
differentiationService.SolveFiniteDiff(request, variant);
        return Ok(response);
    }

    // ...
}
```

кінець лістингу 3.7

Основне завдання чисельного інтегрування полягає в пошуку наближеного значення визначеного інтеграла на заданих проміжках. Як і у випадку з диференціюванням, чисельні алгоритми тут використовуються, якщо:

– аналітичний метод вимагає занадто багато перетворень;

– підінтегральна функція виражена через масив точок.

Варто згадати, що геометричним змістом інтеграла є площа фігури, обмежена графіком функції, віссю абсцис та вказаними межами. Чисельні методи змінюють обрахунок цієї площі сумою площ таких фігур: прямокутників, трапецій або парабол.

На рисунку 3.8 наведено діаграми класів, що використовуються в межах чисельного інтегрування.

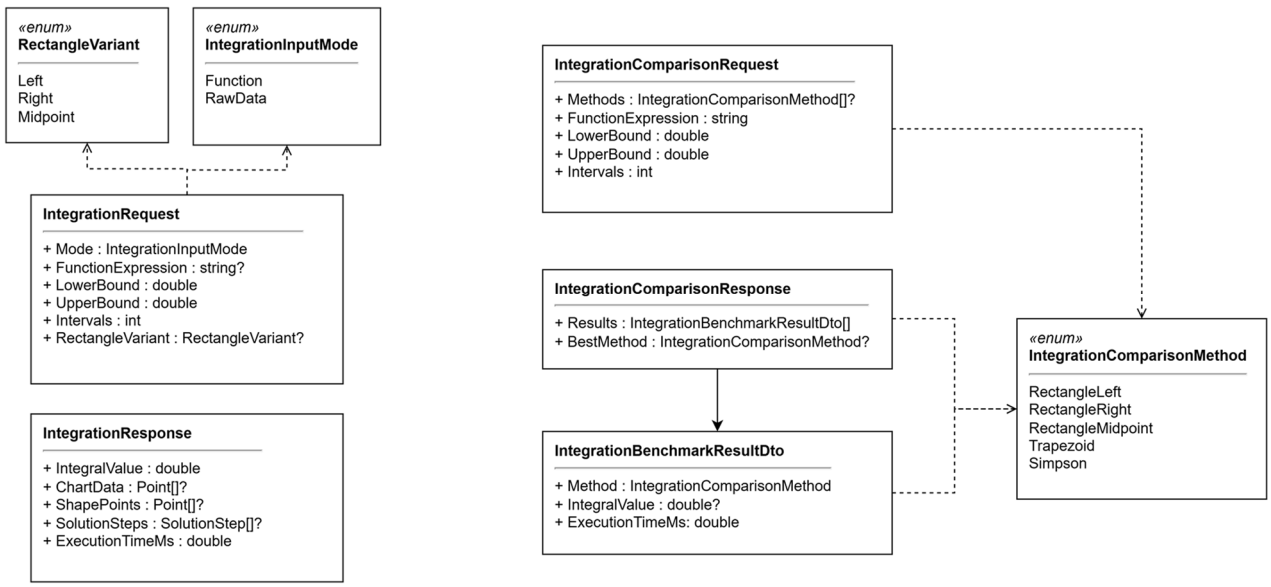


Рисунок 3.8 – DTO-моделі для чисельного інтегрування

У випадку із методом Сімпсона, який вимагає для обрахунків парну кількість розбиттів, API автоматично перевизначає передану властивість на більше парне число.

Задача оптимізації чисельними алгоритмами полягає у визначенні екстремумів функції (її мінімуму та максимуму) у вказаній області з допустимою похибкою. Система NumCalc дозволяє використовувати для цієї задачі наступні три методи:

– метод рівномірного пошуку, що також ще називають методом сітки;

– метод золотого перерізу з високою швидкістю збіжності;

– метод градієнтного спуску або підйому, що може виконуватися і для багатовимірних функцій. Під час його виклику Calculation API повертає масиви точок, необхідні для побудови графіка (зокрема на 3D-поверхні).

Нижче, на рисунку 3.9, наведено моделі, що побутують в домені оптимізації.

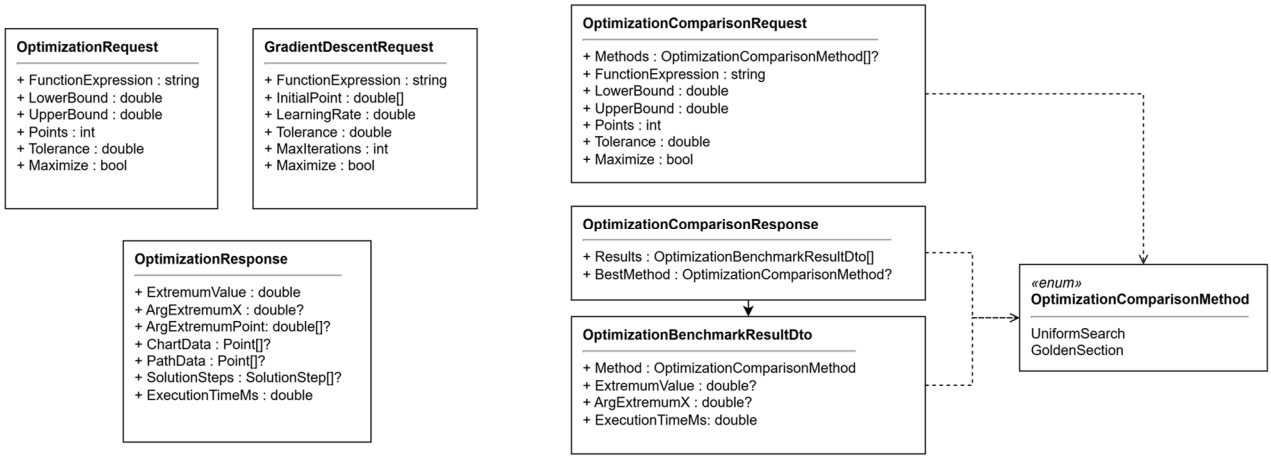


Рисунок 3.9 – DTO-моделі, пов’язані з оптимізацією

Властивість Points в Request-моделі позначає кількість вузлів для рівномірного пошуку. Тому дана властивість не стосується методів «золотого перетину» та градієнтного спуску/підйому. Модель відповіді сервера містить наступні властивості:

– ExtremumValue – віднайдене значення мінімуму або максимуму функції, заради якого й виконувався алгоритм;

– ArgExtremumX – координата, в якій функція має екстремум;

– ArgExtremumPoint – вектор координат для багатовимірних функцій. ArgExtremumPoint стосується тільки градієнтного методу;

– PathData – траєкторія ітерацій, по якому алгоритм градієнтного спуску/підйому збігався. Так само стосується тільки градієнтного методу.

Вибір, що саме варто шукати методу – максимум чи мінімум, ґрунтується на параметрі булевого значення – Maximize.

Остання задача, з вирішенням якої може допомогти Calculation API – розв’язок звичайних диференціальних рівнянь. Задача Коші для ЗДР полягає у віднайденні такої функції, яка б змогла задовольнити диференціальне рівняння та початкову умову, зазначені у формулі 3.2.

$$y' = f(x, y), y(x_0) = y_0. \tag{3.2}$$

В реальності отримати точний аналітичний розв’язок ЗДР майже неможливо аналітично. Проте побудована система дозволяє користувачеві вирішувати дане завдання з використанням наступних чисельних алгоритмів:

– метод Ейлера з першим порядком точності;

– метод Гойна (удосконалений метод Ейлера) з другим порядком точності;

– метод Рунге-Кутта 2-го порядку;

– метод Рунге-Кутта 4-го порядку, що забезпечує 4-ий порядок точності;

– метод Пікара, що будує розв’язок у вигляді степенового полінома.

Ознайомимося з моделями, що фігурують відносно розв’язків звичайних диференційних рівнянь (рис. 3.10).

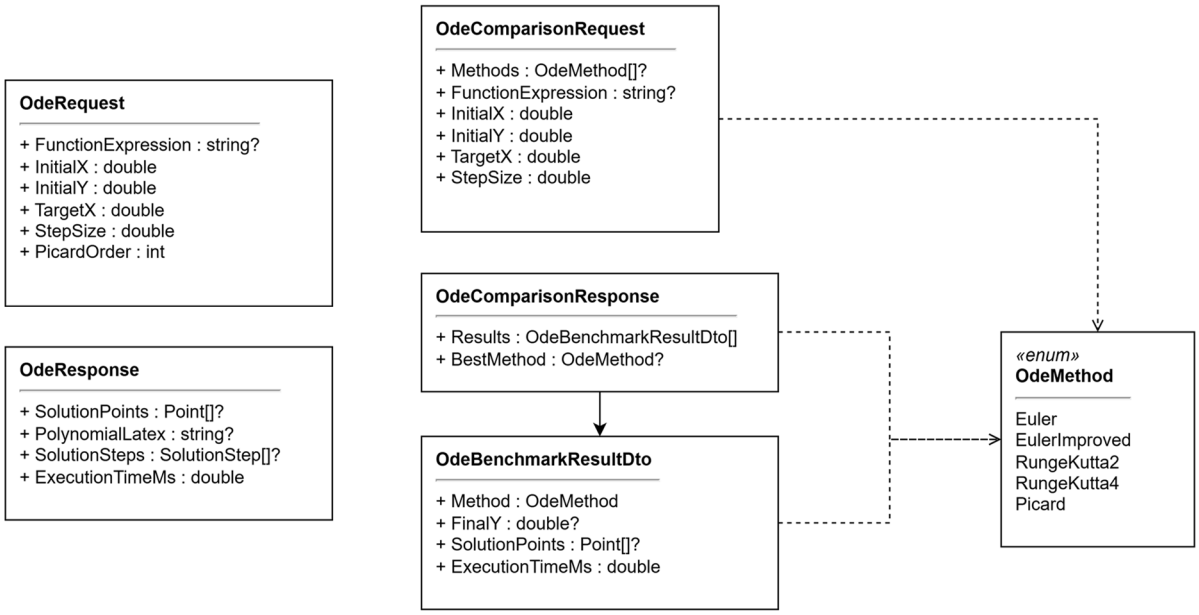


Рисунок 3.10 – DTO-моделі, пов’язані з розв’язанням ЗДР

В OdeRequest права частина рівняння передається, як FunctionExpression. Така властивість, як PicardOrder, відповідає за порядок наближення, може набувати значення від 1 до 10, ігнорується іншими методами. В OdeComparisonRequest даний PicardOrder не включається, оскільки метод Пікара не доступний для порівняння на рівні з іншими чисельними методами. Було вирішено виключити даний алгоритм з можливості порівняння, оскільки він є більше символьно-аналітичним, а не чисельним. Властивість PolynomialLatex також стосується тільки методу Пікара.

Таким чином, всі реалізовані чисельні та, у деяких типах задач, аналітичні методи дозволяють майбутнім користувачам швидко та зручно виконувати обчислення на стороні серверної інфраструктури через взаємодію з UI-системою у формі вебсайту або застосунків. Всі алгоритми та формули узгоджені з існуючими науковими напрацюваннями [5, 17].

Останнім модулем Calculation API на огляд є OCR імплементація, що відповідає за надсилання масиву байтів зображення на обробку певному іншому зовнішньому сервісу задля розпізнавання математичних виразів. Для цього було додано:

– OcrController. Містить один POST-ендпойнт – «/recognize», який з тіла запиту дістає модель OcrRequest – обгортку навколо рядкового значення, що відображає зображення у формі Base64-кодування. Якщо результат успішно оброблено, то клієнтській частині повертатиметься модель OcrResponse з властивістю Latex;

– OcrService. Частина загальної бізнес-логіки проєкту NumCalc.Calculation.Business. Даний сервіс, що імплементує метод RecognizeAsync з інтерфейсу IOcrService, підключає визначений OCR-провайдер (IOcrProvider), та передає йому витягнене зображення в Base64-кодуванні та його MIME-тип. MIME-тип – стандарт, що визначає формат файлу, який передається по інтернет-мережі. Він має бути вкладений у моделі запиту, як частина OcrRequest (загальна схема – «data:{MIME-тип};base64,{дані}»);

– IOcrProvider. Інтерфейс, контракт з яким вимагає реалізацію методу

RecognizeAsync. Якщо з’явиться якийсь новий сервіс, який зможе здійснити витягування математичного виразу із зображення, то достатньо буде додати новий провайдер і замінити в «Program.cs» імплементацію з поточної на нову. Наразі, поточним OCR-провайдером є сервіс GeminiOcrProvider, який звертається до Google API інфраструктури. В лістингу Г.2 можна ознайомитися із структурою даного сервісу.

Провайдер має такі константи та поля:

– Endpoint. Константа, яка вказує на те, куди ми маємо звернутися при надсиланні запиту по HTTP;

– Prompt. Запит, який передаватиметься генеративній моделі, в якому чітко вказано, який результат має повернути модель;

– _apiKey. API-ключ, який використовується для ідентифікації особи (в даному випадку сервісу), що здійснює запит до Google API. Отримати його можна в Google AI Studio в межах визначеного Google-акаунту. Оскільки такі дані є чутливими, то даний ключ зберігається в .env-файлі (локальна розробка) та environment variables на стороні контейнера сервісу в Azure (розгорнута інфраструктура для користувачів);

– JsonOptions. Опції, які має JSON-серіалізатор враховувати під час десеріалізації отриманої відповіді зі сторони Google API.

Оскільки структуру проєктів NumCalc.Calculation.Api та NumCalc.Calculation.Business повністю описано, можна ознайомитися із серверною частиною, пов’язаною з користувачами – NumCalc.User. Зважаючи на те, що дана частина проєкту реалізована за таким архітектурним шаблоном, як Clean Architecture, то в її межах було створено 4 C#-проєкти:

– NumCalc.User.Domain. Фундамент проєкту, в якому знаходяться сутності з бази та перелічення (CalculationType та UserErrorCode). Він не має жодних залежностей, тому принцип Clean Architecture тут дотримано повністю;

– NumCalc.User.Application. Містить інтерфейси для репозиторіїв (Data Access Layer, DAL) та сервісів бізнес-логіки, і DTO-моделі, що будуть вертатися як відповіді від сервера. Таким чином сутності будуть репрезентувати суто

табличні дані;

– NumCalc.User.Infrastructure. Включає в себе реалізації всіх інтерфейсів з NumCalc.User.Application та конфігурацію контексту бази даних;

– NumCalc.User.API. Презентаційний шар побудованої архітектури, де знаходяться контролери і middlewares (проміжне програмне забезпечення). Також тут зберігаються файли із логами, записаними під час виконання системи.

На відміну від API обчислень дана підсистема не має залежності від NumCalc.Shared, задля уникнення тої ситуації, що NumCalc.User.Application мусив би мати залежність від зовнішньої бібліотеки, що порушувало б Clean Architecture. Загалом архітектуру можна зобразити так, як це показано на рисунку 3.11.

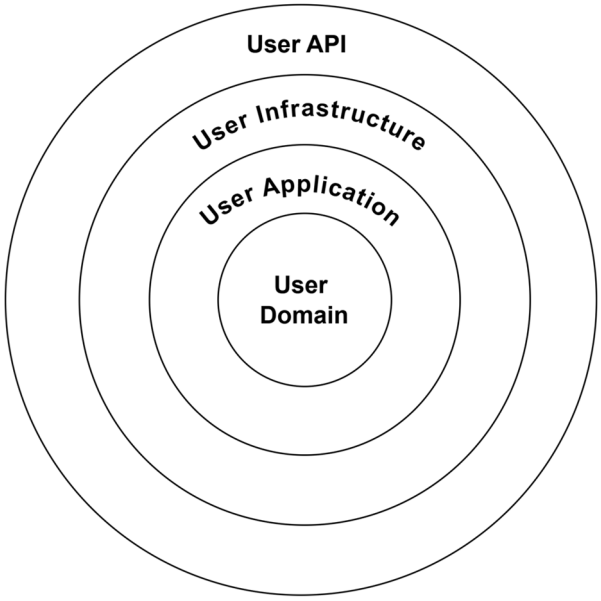


Рисунок 3.11 – Архітектура користувацького API

NumCalc.User.Domain вміщує наступні сутності:

– AppUser – користувач;

– CalculationHistoryRecord – запис історії про певний обрахунок;

– PasswordResetToken – інформація про конкретний токен на відновлення паролю;

– SavedFile – сутність, що відображає інформацію про конкретний

згенерований PDF-рапорт по якомусь обрахунку;

– SavedInput – збережені користувачем дані, введені ним же самим;

– BaseEntity – абстрактний клас, який має властивості, спільні для усіх сутностей: ідентифікатор (Id) та дата створення запису (CreatedAt).

Перелічення, що розміщені в цьому C#-проєкті також, потрібні або для сутностей (CalculationType), або для чіткого опису помилки, що виникала під час виконання методів на стороні BLL-сервісів (UserErrorCode).

Інтерфейси, що є в NumCalc.User.Application, погруповані по підпапках: «Repositories» та «Services». Важливими інтерфейсами сервісів, мета яких не пов’язана чисто з взаємодією з репозиторіями, є: IAuthService, IEmailSender та IJwtService. IAuthService вимагає реалізації методів відносно реєстрації (RegisterAsync), логіну (LoginAsync), оновлення паролю (ResetPasswordAsync та RequestPasswordResetAsync). Тим часом IEmailSender уособлює провайдер з надсилання електронних листів, а IJwtService – генератор токенів (в нашій системі поки тільки access-токен). Refresh-токен загалом в проєкті не фігурує, оскільки не було великої необхідності в розробці функціоналу, який не дуже важливий в подібних системах. Він би створював тільки зайвий клопіт.

Розглянемо NumCalc.User.Infrastructure, що реалізовує основний бізнес-функціонал системи. В папці «Configuration» зберігаються моделі налаштувань, які зберігаються або в «appsettings.json», або в .env-файлах, або в змінних середовища. Сюди відносяться налаштування необхідні для підключення до SMTP-сервера Gmail (ліст. 3.8) та URLs до середовищ.

Лістинг 3.8 – Налаштування для зв’язку з SMTP-сервером

```
public class SmtpSettings
{
    public string Host { get; set; } = string.Empty;
    public int Port { get; set; }
    public string Username { get; set; } = string.Empty;
    public string Password { get; set; } = string.Empty;
    public string FromAddress { get; set; } = string.Empty;
    public string FromName { get; set; } = string.Empty;
    public bool UseSsl { get; set; }
}
```

кінець лістингу 3.8

Опис контексту бази даних наведено в лістингу 3.9.

Лістинг 3.9 – Опис контексту бази даних

```
public class AppDbContext(DbContextOptions<AppDbContext>
options) : DbContext(options)
{
    public DbSet<AppUser> Users { get; set; }
    public DbSet<CalculationHistoryRecord>
CalculationHistoryRecords { get; set; }
    public DbSet<SavedInput> SavedInputs { get; set; }
    public DbSet<SavedFile> SavedFiles { get; set; }
    public DbSet<PasswordResetToken> PasswordResetTokens { get;
set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {

modelBuilder.ApplyConfigurationsFromAssembly(typeof(AppDbContext).
Assembly);
        base.OnModelCreating(modelBuilder);
    }
}
```

кінець лістингу 3.9

Даний контекст успадковується від DbContext та дозволяє, на основі прописаних конфігурацій в папці «Configurations», створити необхідні таблиці та прописати їхнім колонкам правила. Лістинг 3.10 відображає приклад такої конфігурації.

Лістинг 3.10 – Опис конфігурації сутності користувача

```
public class AppUserConfiguration :
IEntityTypeConfiguration<AppUser>
{
    public void Configure(EntityTypeBuilder<AppUser> builder)
    {
        builder.ToTable("Users");

        builder.HasKey(u => u.Id);
        builder.Property(u => u.Id)
            .ValueGeneratedOnAdd();

        builder.Property(u => u.Username)
            .IsRequired()
            .HasMaxLength(50);
```

```
        builder.HasIndex(u => u.Username).IsUnique();
        builder.Property(u => u.Email)
            .IsRequired()
            .HasMaxLength(256);
        builder.HasIndex(u => u.Email).IsUnique();
        builder.Property(u => u.PasswordHash)
            .IsRequired();
        builder.Property(u => u.CreatedAt)
            .IsRequired()
            .HasDefaultValueSql("GETUTCDATE()");
    }
}
```

кінець лістингу 3.10

На основі доданих конфігурацій, .NET генерує шаблон для бази – AppDbContextModelSnapshot, який має бути застосований при публікації проєкту. Наступні внесені зміни в базі, які називаються міграціями, також створюють окремі файли із описом того, що потрібно застосувати до структури бази та її таблиць.

На лістингу 3.11 представлено приклад створеного репозиторію, що приймає в конструктор зареєстрований в життєвий цикл API-застосунку AppDBContext.

Лістинг 3.11 – Приклад репозиторію

```
public class PasswordResetTokenRepository(AppDbContext
dbContext) : IPasswordResetTokenRepository
{
    public Task<PasswordResetToken?> GetByHashAsync(string hash,
CancellationToken ct)
    {
        return dbContext.PasswordResetTokens.FirstOrDefaultAsync(t
=> t.TokenHash == hash, ct);
    }

    public Task<PasswordResetToken?> GetByUserIdAsync(Guid userId,
CancellationToken ct)
    {
        return dbContext.PasswordResetTokens.FirstOrDefaultAsync(t
=> t.UserId == userId, ct);
    }

    public async Task AddAsync(PasswordResetToken token)
    {
        await dbContext.PasswordResetTokens.AddAsync(token);
```

```
    }

    public void Delete(PasswordResetToken token)
    {
        dbContext.PasswordResetTokens.Remove(token);
    }

    public Task SaveChangesAsync()
    {
        return dbContext.SaveChangesAsync();
    }
}
```

кінець лістингу 3.11

Важливо зазначити, що у всі методи BLL-сервісів та DAL-репозиторіїв передаються cancellation-токени, які використовуються задля скасування операції, результати якої вже не очікуються. Наприклад, в тому разі, коли користувач закрив вкладку зі сторінкою у браузері. Такий підхід дозволяє зменшити навантаження на сервер та зекономити його ресурси. Підхід до функціональності BLL-сервісів застосований такий самий, як і у API обчислень: на випадок невалідності Request-моделей, викидається виняток, який перехоплюється налаштованим глобальним обробником помилок. Проте на відміну від Calculation API, який на випадок перехоплення CustomException вертав тільки 400-ий HTTP статус-код, User API має інакшу модель CustomException, яка також приймає як параметр код (ліст. 3.12).

Лістинг 3.12 – Модель CustomException в API користувачів

```
public class CustomException(UserErrorCode errorCode, string
message, int statusCode) : Exception(message)
{
    public UserErrorCode ErrorCode { get; } = errorCode;
    public int StatusCode { get; } = statusCode;
}
```

кінець лістингу 3.12

NumCalc.User.API містить як і контролери, які не вимагають автентифікації користувача, так і ті, що вимагають. Це запобігає отримати дані користувачів тим людям, які не мають на це жодних прав. ID користувача, дані якого треба

витягнути, дістається з access-токена, що закріплений в заголовку HTTP-запиту. Вся реєстрація логування, сервісів, Swagger-а ітд., здійснюється в межах створення застосунку в «Program.cs». На відміну від системи обрахунків, дана система користувачів також реєструє схему автентифікації з використанням методу AddJwtAuthentication (ліст. 3.13).

Лістинг 3.13 – Реєстрація схеми автентифікації

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

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.TokenValidationParameters = new
TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                ValidIssuer = configuration["JWT:Issuer"],
                ValidAudience = configuration["JWT:Audience"],
                IssuerSigningKey = new SymmetricSecurityKey(

Encoding.UTF8.GetBytes(configuration["JWT:Secret"]!))
            };
        });

    return services;
}
```

кінець лістингу 3.13

Як і у випадку з NumCalc.Calculation.Api, для розгортання середовища в Docker-контейнері на платформі Azure, NumCalc.User.API також має «Dockerfile» для створення image-ів API-застосунку.

Останньою частиною для завершення огляду системи NumCalc є solution-папка NumCalc.UI – сукупність проєктів, призначених для можливості надання доступу користувачам до вирішення математичних задач з використанням Calculation API. Складається вона з:

– NumCalc.UI.MAUI. C#-проєкт кросплатформного застосунку;

– NumCalc.UI.Web. Проєкт, що являється Blazor Web App – вебплатформою;

– NumCalc.UI.Shared. Бібліотека компонентів, сторінок та статичних вебресурсів, що використовуються в NumCalc.UI.MAUI та NumCalc.UI.Web.

Спочатку ознайомимося з бібліотекою компонентів, яка за своєю природою є RCL – Razor Class Library. В нас вона містить:

– компоненти та сторінки;

– resx-файли, що містять локалізаційні ключі та їхні значення для української та англійської мов;

– моделі, специфічні саме для UI;

– макети, які є як для загальних сторінок (MainLayout), так і для сторінок з реєстрацією, логіном тощо;

– HTTP-сервіси, що допомагають комунікувати з внутрішніми API в системі;

– UI-сервіси та статичні допоміжні класи;

– CSS-стилі, шрифти, зображення, JS-скрипти тощо.

Колірна палітра підібрана з метою гармонійного поєднання кольорів з малиновим відтінком (рис. 3.12).

Рисунок 3.12 – Колірна палітра NumCalc

На початку планувалося використання таких сімейств шрифтів: Space Mono та Mulish. Перший підходить для відображення заголовків, другий – для звичайного текстового наповнення. Обидва з них гарно комбінуються із

загальною тематикою системи. Проте під час розробки було виявлено, що Space Mono не підтримує кириличні символи, тому замість нього було обрано – IBM Plex Mono, який максимально подібний до попереднього. Всі шрифти безкоштовні та були доступні для завантаження в інтернеті.

Серед компонентів є модальні вікна (BaseModal), панелі (ResultPanel), таблиці (NodeTable), розділювачі (Divider), обгортки навколо SVG-іконок (SvgIcon), підказки (Tooltip) тощо. Деякі з них пов’язані тільки зі сторінками для обчислень, деякі інші – зі сторінками для роботи користувацького функціоналу. Є компоненти, що дозволяють винести специфічну важку логіку зі сторінок: як у випадку із формами для заповнення початкових даних. Особливо специфічним компонентом, який мало де використовується на інших сайтах, є LatexDisplay, що дозволяє відмальовувати різні математичні вирази, записані в LaTeX-форматі, читабельно з використанням такої JS-бібліотеки, як katex. Його кодову частину наведено в лістингу 3.14, як приклад того, як виглядає code-behind в Blazor.

Лістинг 3.14 – Кодова частина компонента LatexDisplay

```
public partial class LatexDisplay : ComponentBase
{
    [Parameter] public string? Latex { get; set; }
    [Parameter] public string? Prefix { get; set; }
    [Parameter] public string? AdditionalCssClass { get; set; }
    [Inject] private IJSRuntime JsRuntime { get; set; } = null!;
    private readonly string _containerId = $"latex-display-
{Guid.NewGuid():N}";
    private bool _rendered;
    protected override void OnParametersSet() => _rendered =
false;
    protected override async Task OnAfterRenderAsync(bool
firstRender)
    {
        if (_rendered || string.IsNullOrEmpty(Latex)) return;
        _rendered = true;
        await
JsRuntime.InvokeVoidAsync("LatexHelper.renderLatexById",
_containerId, Prefix + Latex);
    }
}
```

кінець лістингу 3.14

Найбільш часто використовуваними компонентами є Dropdown (з можливістю вибору тільки одного або багатьох значень) та Switch. Також незвичним компонентом є OcrInput, що дозволяє отримати готовий математичний вираз, відображений на зображенні, у LaTeX-записі. Являється звичайною кнопкою з іконкою зображення, розташованою всередині всіх полів вводу функцій. Даний компонент експлуатує два модальні вікна: один з кнопкою завантаження файлу-зображення, а інший для обробки зображення перед його відправкою на API обчислень на відповідний ендпойнт. Останнє згадане модальне вікно є також окремим компонентом – CropImageModal. Обробка зображення здійснюється з використанням такого NuGet-пакета, як Cropper. На рисунку 3.13 можна побачити візуальний вигляд модалки.

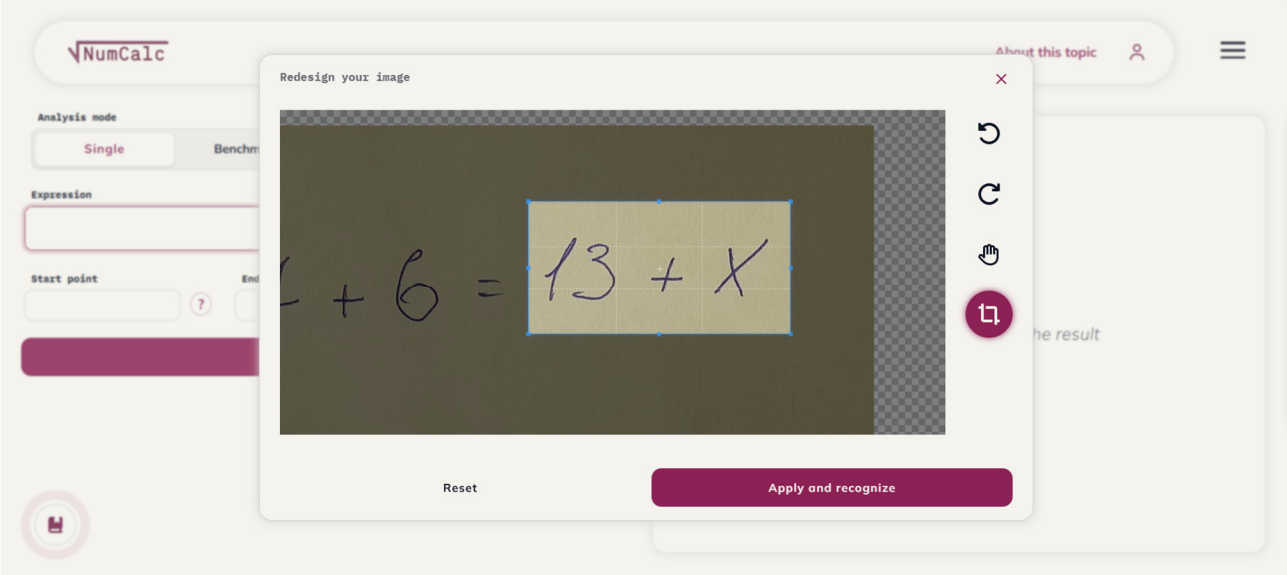


Рисунок 3.13 – Візуальний вигляд CropImageModal

Сторінок, які за своєю суттю теж є Blazor-компонентами, але з атрибутом @page, теж чимало. Всі з них є дочірніми класами відносно або BasePage, або AuthorizedPage (який також є дочірнім класом BasePage, але зі своєю логікою). BasePage – абстрактний клас, який містить метод-обгортку для безпечного виконання запитів до API, різні підключені сервіси тощо. AuthorizedPage необхідні для сторінок, функціонал до яких доступний тільки авторизованим користувачам. В межах бібліотеки було створено наступні сторінки:

– по одній на кожну математичну задачу;

– головна (MainPage);

– для логіну (Login), реєстрації (Register), запиту на відновлення паролю (ForgotPassword) та безпосереднє його відновлення (ResetPassword);

– короткий підсумок по активності користувача (UserDashboard), налаштування акаунту (AccountSettings);

– для перегляду історії обрахунків (CalculationHistory), останніх завантажених результатів (SavedFiles) та збережених введених даних на обчислювальних сторінках (SavedInputs).

Кожна зі сторінок, на яких вирішуються математичні задачі, має форму для заповнення, графік, контейнер для того аби показувати здійснені операції в межах виконання алгоритму. На рисунку 3.14 можна побачити приклад зі сторінкою для пошуку коренів рівняння.

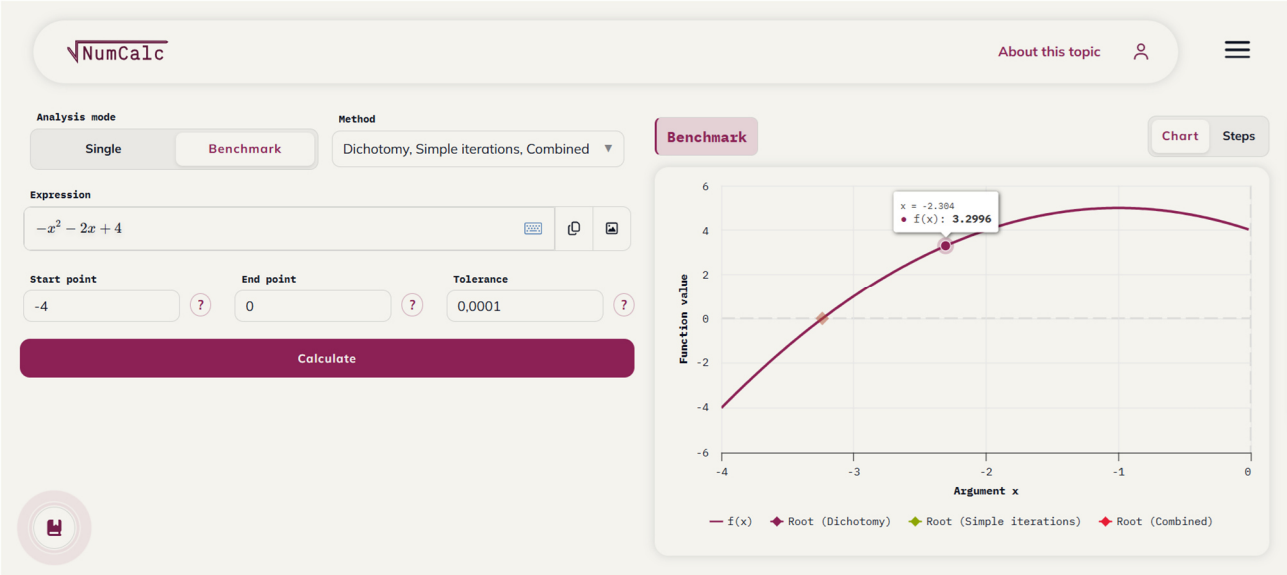


Рисунок 3.14 – Вигляд сторінки для пошуку коренів рівняння

Головна сторінка містить коротку інформацію про систему, кількість реалізованих методів, посилання на репозиторій проєкту в GitHub та електронну пошту автора для можливості зворотного зв’язку. Сторінки логіну та реєстрації мають спільні поля для вводу: користувацьке ім’я та пароль. У випадку реєстрації ще також є поле для підтвердження паролю та для вводу електронної пошти.

Сторінка із відновленням забутого паролю пропонує ввести електронну пошту. Якщо електронна пошта була неправильно вказана, то користувачу не буде відображатися повідомлення про помилку. Це дозволить уникнути потенційних кібератак, пов’язаних зі збором даних про наявних користувачів в системі. Посилання на відновлення паролю отримується з надісланого електронного листа. Налаштування акаунту включають в себе: зміна електронної пошти, імені, паролю та видалення акаунту. Будь-яка зміна вимагає підтвердження паролем. Короткий підсумок активності користувача, що доступна за шляхом «/dashboard», має 3 списки, в яких відображено по 5 пунктів на історію обрахунків, останні завантажені файли та останні збережені введені параметри для обчислень. Кожен з цих списків дозволяє перейти на відповідні сторінки, на яких можна переглянути всю доступну інформацію та видалити непотрібні записи.

Макети – це обгортки для сторінок, які дозволяють винести компоненти, що постійно могли дублюватися на сторінках. RCL має такі: головний (MainLayout) та автентифікаційний (AuthLayout). Головний макет використовує наступні компоненти: «шапка» (Header), меню для перемикання на якусь «математичну» сторінку, контейнер для відображення загальної інформації про якусь «математичну» сторінку (HamburgerMenu), індикатор завантаження (GlobalLoader) та контейнер для відображення toast-повідомлень (ToastContainer). У випадку мобільної версії замість «шапки» показується «footer» (BottomNavigation). Автентифікаційний використовується для таких сторінок, як Login, Register, ForgotPassword та ResetPassword, і містить тільки індикатор завантаження та toast-контейнер (з переліченого в головному макеті). Окрім цих двох компонентів тут також є кнопка «Продовжити як гість», що дозволяє користуватися системою без необхідності в реєстрації (хоча й з деякими обмеженнями).

CSS-стилі написані з використанням БЕМ-методології, що дозволяє структурувати класи та робити «гніздування». Для обробки PostCSS використовується додаткова конфігурація, вміст якої наведено в лістингу 3.15.

Лістинг 3.15 – Конфігурація PostCSS у Vite

```
export default {
    plugins: {
        'postcss-import': {},
        'postcss-nested': {},
        'postcss-custom-media': {},
        'postcss-preset-env': {
            stage: 1,
            features: {
                'custom-properties': false
            }
        },
    }
}
```

кінець лістингу 3.15

В «package.json» вказані всі залежності, які використовуються як під час розробки (devDependencies), так і для забезпечення коректності роботи функціоналу (dependencies) (ліст. 3.16).

Лістинг 3.16 – Конфігурація «package.json»

```
{
  "name": "numcalc.ui.shared",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "watch": "vite build --watch",
    "build": "vite build"
  },
  "keywords": [],
  "license": "ISC",
  "devDependencies": {
    "fast-glob": "^3.3.3",
    "postcss": "^8.5.6",
    "postcss-custom-media": "^12.0.1",
    "postcss-import": "^16.1.1",
    "postcss-nested": "^7.0.2",
    "postcss-preset-env": "^11.1.1",
    "vite": "^7.3.1",
    "vite-plugin-image-optimizer": "^2.0.3",
    "vite-plugin-static-copy": "^4.1.0",
    "vite-plugin-svg-icons": "^2.0.1"
  },
  "dependencies": {
    "highcharts": "^12.5.0",
```

```
    "html2canvas": "^1.4.1",
    "katex": "^0.16.45",
    "mathjs": "^15.1.0",
    "mathlive": "^0.108.2"
  }
}
```

кінець лістингу 3.16

Vite дозволив максимально автоматизувати оновлення стилів та скриптів на льоту, мініфікувати файли, проставляти вендорні префікси до CSS-властивостей тощо. Окрім команди «dev», яка використовувалася під час розробки, також є команда «build», що автоматично виконується при створенні нового image-а вебдодатку («Dockerfile» у NumCalc.UI.Web також є написаний). Написані JS-скрипти, що об’єднуються в один загальний файл – «app.js», вирішують наступні задачі:

– рендер графіків з використанням highcharts, включно із забезпеченням їхнього адекватного завершення життєвого циклу («chart-helper.js»);

– отримання обробленого зображення з компоненту в CropImageModal («image-helper.js»);

– рендер тексту, записаному в LaTeX-форматі («latex-helper.js»);

– перевірку валідності введених математичних виразів («math-helper.js»);

– коректну функціональність поля для вводу функцій, зокрема відкриття та закриття пов’язаної з ним клавіатури («math-input.js»);

– завантаження рапорту про обрахунки: витягування необхідних зображень покрокових етапів та графіків («pdf-helper.js»);

– зміна колірної теми («theme-helper.js»);

– рендер підказок («tooltip-helper.js»).

На рисунку 3.15 можна побачити інтерфейс головної сторінки із застосованою темною темою:

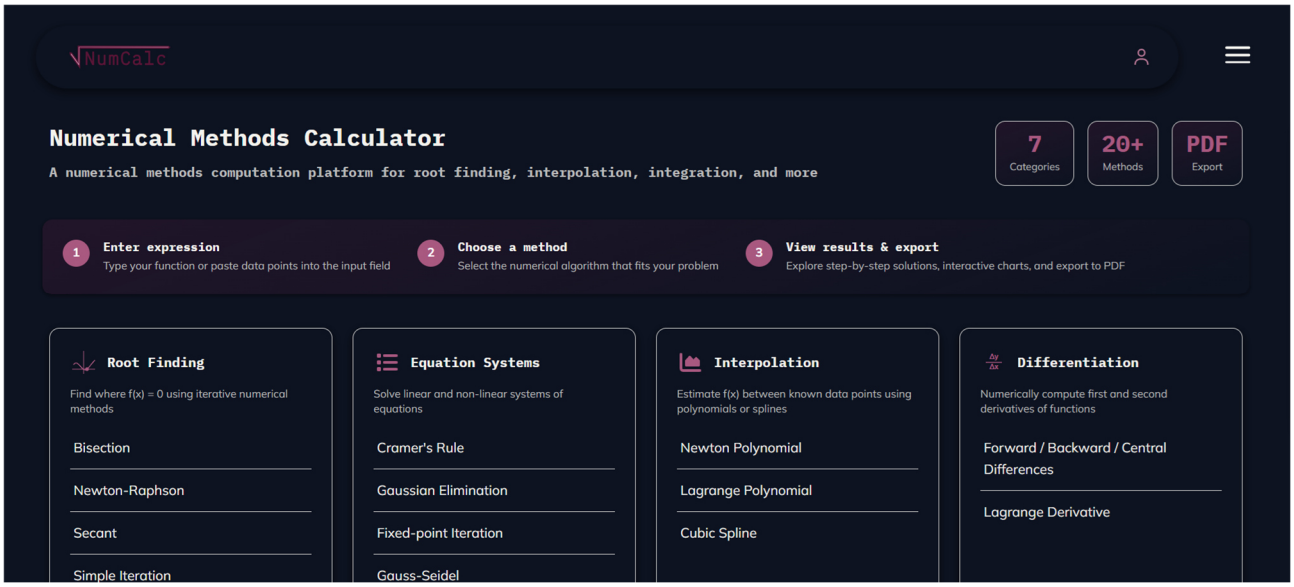


Рисунок 3.15 – Інтерфейс головного екрану з темною темою

RCL має декілька інтерфейсів для UI-сервісів:

– IAuthStateService. Відповідає за управління станом автентифікації користувача в сесії. Сервіс реалізований на стороні NumCalc.UI.Shared;

– ICultureService. Дозволяє дізнатися поточну культуру браузера або системи (у випадку з MAUI) та дозволяє задавати обрану локалізацію. NumCalc.UI.MAUI та NumCalc.UI.Web мають окремі імплементації даного інтерфейсу: застосунок на нативних платформах зберігає дані з використанням API від .NET – Preferences, а в браузері – з localStorage;

– IPdfExportService. Формує файл для завантаження результатів обчислень з використанням QuestPDF;

– IStorageService. Його сервіси відповідають за зберігання даних або в localStorage (вебверсія застосунку), або на пристрої (нативні додатки);

– ITokenStorage. Маніпулює access-токенами в системі: очищує, витягує, зберігає. Реалізації працюють так само, як і в IStorageService;

– IUiStateService. Інтерфейс головного сервісу, що показує індикатор завантаження, toast-повідомлення, інформацію про математичну задачу, коли це необхідно. Загалом керує станом системи. Написаний інтерфейс відображено в лістингу 3.17.

Лістинг 3.17 – Інтерфейс IUiStateService

```
public interface IUiStateService
{
    event Action<bool> OnLoaderChanged;
    event Action<ToastMessage> OnShowToast;
    event Action OnCloseDropdownRequested;
    event Action OnNavMenuChanged;
    event Action<NavigationItem> OnTopicInfoRequested;

    bool IsNavMenuOpen { get; }

    void ShowLoader();
    void HideLoader();

    void ShowError(string message, string title = "Error");
    void ShowSuccess(string message, string title = "Success");

    void RequestCloseDropdown();
    void ToggleNavMenu();
    void CloseNavMenu();
    void RequestTopicInfo(NavigationItem item);
}
```

кінець лістингу 3.17

Дані, які будуть зберігатися в localStorage або в пристрої з використанням Preferences, шифруються. .NET бібліотека Preferences робить це самостійно, зберігаючи ключ шифрування на пристрої. В разі вебсервісу це здійснюється з використанням окремого NuGet-пакета – ProtectedBrowserStorage.

Серед допоміжних extension-класів, на рівні C#-коду, є такі екземпляри:

– ChartUtils для допомоги з перевикористанням якихось додаткових ліній на графіку;

– ColorUtils, що описує колірні CSS-властивості, які лежать в «variables.css». Дозволяє зручно вказувати колір на рівні C#-коду, зменшуючи шанс на можливу помилку в назві;

– MathUtils, який містить методи для форматування чисел та вказання кількості знаків після коми, яку слід відображати;

– NavigationUtils, що має колекцію з елементів навігації.

Також варто зазначити, що в межах рефакторингу сторінок з обрахунками було додано окремі «хелпери» для кожної, з метою покращення читабельності

кодової частини даних Blazor-компонентів. Всі HTTP-сервіси успадковуються або від BaseApiService (якщо не вимагається при зверненні до API автентифікації), або BaseUserApiService (якщо в header запиту потрібно додати поле з токеном доступу). Налаштування логування помилок та подій відрізняється для вебсервісу та нативних застосунків: різниця інфраструктури виконання коду. Підключення UI-сервісів – не відрізняються.

Розгортання всіх сервісів включало використання Docker-контейнерів на стороні Azure інфраструктури. Для цього було створено окремий «Dockerfile» під кожен сервіс: вебсайт (NumCalc.UI.Web), API для обчислень (NumCalc.Calculation.Api) та для користувацьких даних (NumCalc.User.API). Для розгортання було вибрано саму Azure з-поміж інших систем, оскільки в автора був вже досвід роботи з ним. Наявна підтримка розгортання застосунків на основі контейнерів та можливість розгортання SQL Server бази даних також можна вважати перевагами платформи. Побудований CI/CD-пайплайн працює наступним чином:

– створення нових Docker-image-ів на основі Dockerfile в кожному сервісі;

– зберігання нових образів в GHCR (GitHub Container Registry);

– розгортання контейнерів в інфраструктурі Azure після попереднього входження в систему. Дані для логіну під окремий обліковий запис в Azure зберігаються як секретний ключ в GitHub-системі.

Збірка нативних застосунків для Android та Windows відбувається вручну з використанням команди «dotnet publish ...» з явним вказанням цільової платформи. Після її виконання для Android OS буде створено пакети у форматі .apk та .aab, для Windows – пакет MSIX. URL-адреси серверних API знаходяться в статичному класі MauiSettings, тому вони автоматично підставляються при публікації застосунків. Хоча проведення всіх обрахунків на стороні окремого API створює залежність від інтернет-з’єднання, проте це значно зменшує розмір додатків та навантаження на ресурси пристроїв.

3.2 Тестування та налагодження системи для проведення обрахунків чисельними методами

Тестування – один з найважливіших етапів розробки будь-якого програмного комплексу. Відносно побудованої системи NumCalc, тестування включає різноманітні аспекти. Розглянемо тестування Calculation API, яке включає в себе:

– тестування коректності обрахунків усіма чисельними методами, оскільки це є основний функціонал системи;

– загальне навантаження на систему з врахуванням того, що велика кількість одночасних обрахунків може бути тяжкою;

– передача невалідних даних при надсиланні запиту, у відповідь на які потрібно якомога швидше кидати відповідь, ще до моменту використання Python-середовища.

Тестування User API також вимагає перевірку на можливе навантаження системи або проведення фейкових кібератак з метою віднайдення та викрадення якомога більше інформації про наявних користувачів.

Перевірка може здійснюватися як з використанням Postman, так і зі Swagger. У випадку Swagger, на основі XML-коментарів відносно ендпойнтів, контролерів та моделей, автоматично генерується документація, яка дозволяє краще зрозуміти очікувану роботу системи. Під час локальної розробки даного API найчастіше використовувався саме Swagger, який дозволяв зручно бачити наявні контролери та ендпойнти в підсистемі. Даний інструмент також був застосований під час тестування API користувачів.

Як можливе вдосконалення системи, варто було б в майбутньому всі BLL-сервіси та Python-методи покрити юніт-тестами, що дозволило б надати більше впевненості в тому, що функціонал не буде поламаний при якихось внесених змінах. Оскільки .NET та Python екосистеми – різні, то й бібліотеки також різні використовувалися б. Для .NET найпопулярнішою бібліотекою є xUnit, а в Python – pytest. Тести Python-методів хоч і теж можна проводити в проєкті, проте

розміщувати їх варто поза межами папки «Scripts», оскільки CSnakes не має їх враховувати при формуванні інтерфейсів.

Тестування UI-системи включає перевірку:

– коректності роботи в разі введення невалідних даних в поля форм;

– зрозумілого відображення отриманих результатів з API;

– адаптації UI на різних платформах та девайсах.

Воно може здійснюватися як мануально, так і автоматизовано з використанням таких бібліотек, як Selenium або bUnit. Під час розробки активно використовувався такий інструмент – Chrome DevTools, Його вистачало для того, аби швидко перевірити вигляд сторінок та компонентів на великих та малих екранах, аналізувати запити по мережі, перегляди наявні статичні ресурси на клієнтській стороні тощо. Важливим аспектом було використання можливості в Git створювати нові, окремі від main, гілки. Це дозволяло впевнено проводити рефакторинг системи, адже будь-який розробник впевненіше почувається, коли розуміє, що внесені зміни за будь-яких обставин поки що не зачеплять основну гілку з робочим функціоналом.

Тестування безпосередньо кросплатформних систем також є важливим пунктом, оскільки навіть версія операційної системи може вплинути на роботу платформи.

Отже загалом система NumCalc має:

– коректно функціонувати у випадку надсилання невалідних даних;

– витримувати високі навантаження;

– не давати можливості дізнатися кіберзлочинцям про потенційні дані користувачів;

– мати робочий функціонал навіть тоді, коли зовнішні сервіси, такі як Google Gemini API, недоступні. Пов’язаний з якимось недоступним зовнішнім ресурсом функціонал, звичайно, може не працювати. Але загалом це не має впливати на працездатність всієї системи.

3.3 Інструкція користувача з використання програмного продукту

Основною метою розробки було створення програмного забезпечення для користувачів. Будь-яка людина може в декілька способів скористатися NumCalc: або через розгорнутий вебсайт або через інсталятори нативних застосунків.

Вебсайт є постійно доступним за умови наявності інтернет-з’єднання та коректної роботи платформи Azure. Для проведення обрахунків не обов’язково реєструватися в системі.

Інсталятори можна отримати або попросивши їх в розробника цієї системи, або створивши їх самостійно. В останньому випадку достатньо стягнути публічний GitHub-репозиторій (назва – NumCalc, автор – klouzhenk), та в корені проєкту виконати наступні команди:

1) «npm ci --prefix NumCalc.UI.Shared»;

2) «npm run build --prefix NumCalc.UI.Shared»;

3) «dotnet workload install maui»;

4) «dotnet restore NumCalc.sln»;

5) «dotnet publish NumCalc.UI.MAUI -f net9.0-android -c Release» або «dotnet publish NumCalc.UI.MAUI -f net9.0-windows10.0.19041.0 -c Release».

Після їхнього виконання, готові пакети з’являться в папці «bin/Release/{цільова-платформа}/publish».

Якщо користувач хоче запустити проєкти локально, він може скористатися такою командою, як «docker compose up --build». Дана команда виконує скрипт, описаний в «docker-compose.yml»: завантажує не встановлений image для розгортання бази в контейнері, отримує image двох API та вебсайту, та запускає їх в контейнері, передаючи також для деяких сервісів змінні середовища. Змінні середовища зчитуються з локального файлу «.env». Для отримання необхідних додаткових даних або консультації, можна зв’язатися з автором репозиторію. Якщо User API запуститься тільки, тоді коли БД пройде health check, то Web UI в свою чергу чекає на успішний запуск Calculation API та User API. Головною вимогою для виконання команди є наявність встановленого та запущеного

Docker Engine. Вся npm збірка та встановлення Python-модулів для обчислень виконується автоматично. На кінець в Docker Desktop має бути доступний для перегляду compose stack із активними контейнерами (рис. 3.16).

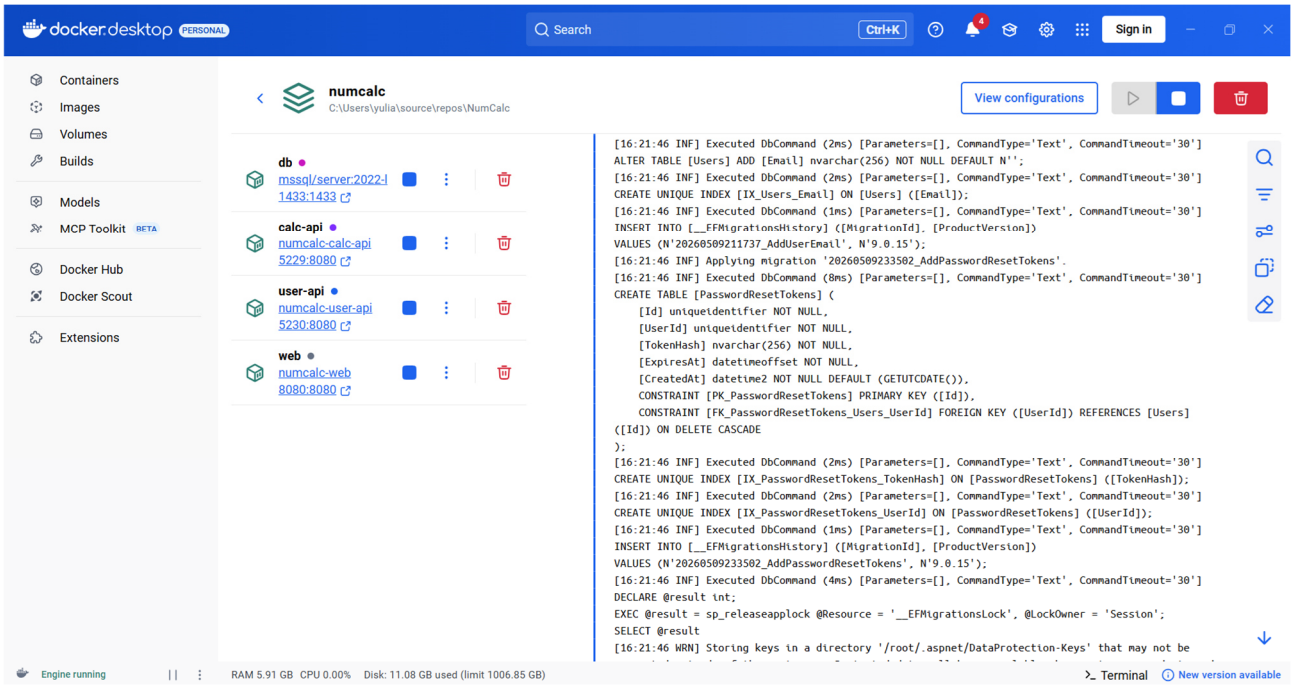


Рисунок 3.16 – Огляд compose stack в Docker Desktop

Дана програма дозволяє зручно переглядати залоговані події та помилки, зупиняти або запускати створені контейнери, таким чином пришвидшуючи швидкість розробки та тестування продукту.

РОЗДІЛ 4
ЕРГОНОМІЧНА ОЦІНКА ФУНКЦІОНАЛЬНИХ КОМПОНЕНТІВ ПРОГРАМНОГО ПРОДУКТУ

Вебчастина проєкту NumCalc побудована таким чином, щоб забезпечити хороший досвід в користуванні продуктом. Всі сторінки, пов’язані з проведенням обчислень, мають спільну структуру: зліва користувач має ввести дані, справа відображатимуться результати обчислень, що створює відчуття певної передбачуваності та надійності в роботі. Біля кожного поля або пункту на UI відображаються маленькі підказки з поясненням того, що яка властивість означає.

Кожна сторінка з обчисленнями має коротке пояснення до методів вирішення конкретних завдань. Шрифти та кольори підібрані таким чином, щоб користувачу було забезпечено достатню контрастність елементів. Колірна гама – передбачувана і містить невелику кількість кольорів, які одне з одним можуть контрастувати. Якщо для помилок та виведення попереджень застосовуються червоний та жовтий відтінки, то для успішних результатів – зелений відтінок. З повагою до користувацьких змінених стилів, зокрема відносно розмірів шрифтів, при написанні CSS-правил використовувалися rem-одиниці, замість пікселів. В межах розробки було реалізовано дві колірні теми: світла та темна, які користувач може явно самостійно собі обрати, без необхідності використання браузерських розширень.

Усі потенційні довготривалі процеси супроводжуються індикаторами завантаження. Відображення графіків та покрокових операцій виконання певного алгоритму допомагає краще зрозуміти або згадати як працює той чи інший метод. Всі потенційні винятки, які могли статися під час виконань якихось операцій на UI, перехоплюються, що запобігає «руйнуванню» застосунків. Наявність підтримки англійської та української локалізації дозволяє охопити більшу цільову аудиторію. Помилки, які вертаються з API можуть бути зручно локалізовані. Верстка сайту є адаптивною: всі компоненти знаходяться на своєму місці незалежно від розмірності екрану. Продуманий UX враховує те, що

користувачу на мобільних пристроях зручніше буде користуватися сайтом та застосунками, якщо основний функціонал для взаємодії буде розміщений знизу екрана (як у випадку з навігацією). Модальні вікна можуть або закриватися по натисканню поза їх межами, або залишатися відкритими, як у випадку з модальним вікном корегування зображення перед відправкою на розпізнавання.

Під час тестування було виявлено на початку декілька невідповідностей:

– незручність розташування елементів на екранах мобільних пристроїв. Габаритна «шапка» займала багато місця на екрані, якого й так небагато;

– тяжке введення чисел в полях для вводу, оскільки при повному стиранні автоматично проставлялися нулі (система не дозволяла ні на секунду залишити поля пустими);

– відсутність перекладів в багатьох місцях;

– неповна інформація в модальних вікнах з інформацією про певний вид обчислень;

– відсутність графіку при презентуванні результатів, яке вирішувалося повним перезавантаженням сторінки;

– занадто великі шрифти на мобільних пристроях.

Якісь проблеми вимагали зміни логіки або її розширення, якісь вирішувалися моментально. Загалом критичні помилки в роботі функціоналу, які можуть негативно вплинути на користувацький досвід – відсутні. Як і в будь-якому програмному забезпеченні, проблеми можуть бути присутні, але тимчасово не віднайдені, через багатий функціонал. Проте, як і за Agile-методологією, розробка програмного забезпечення є постійним ітераційним процесом. Головне – вчасно віднаходити такі помилки в системі та надати можливість користувачам розповісти про свої враження від користування функціоналом.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було побудовано інфраструктуру для надання можливості користувачам розв’язувати задачі з використанням чисельних методів.

Проаналізувавши найпопулярніші застосунки в площині математичних обчислень, було вирішено реалізувати систему з двома API, базою даних та кросплатформним додатком. Було проаналізовано найпопулярніші застосунки відносно математичних обчислень: Photomath, WolframAlpha, mathros, Geogebra та Desmos. На основі визначених переваг та недоліків кожної дослідженої програмн, було вирішено реалізувати систему з двома API, базою даних та кросплатформним додатком.

Для розробки було обрано такий технологічний стек, як .NET, що дозволило реалізувати одну кодову базу для вебсайту та кросплатформних застосунків. Частина даного стеку – ASP.NET Core – було застосовано при розбудові серверної частини проєкту. Для виконання математичних обчислень було обрано NuGet-пакет – CSnakes, що дозволяє викликати Python-код на основі згенерованих бібліотекою C#-інтерфейсів. Екосистема Python надає доступ до найбільш обширних за функціоналом пакетів для проведення обчислень: SymPy, NumPy, SciPy ітд. В межах розробки UI-системи було вирішено застосовувати пакети з Node Package Manager.

Перед початком розробки програмного забезпечення було проаналізовано вимоги до поставленої задачі та продумано архітектуру рішення. Сформульовані функціональні та нефункціональні вимоги до системи надали змогу описати майбутню архітектуру з використанням C4-методології, sequence та ER-діаграм.

Серверна частина складається з двох частин: API для виконання обрахунків та API для доступу до користувацьких даних. Основний функціонал реалізовано в першій частині із застосуванням ASP.NET Core, CSnakes та функцій, написаних на Python. Користувацьке API є більш подібним до типових CRUD-систем в .NET. Якщо Calculation API представляє N-Layered Architecture, то другий – Clean

Architecture. Інтеграція із зовнішніми сервісами: Google Gemini API та Gmail SMTP Server, налаштована успішно.

Завершилася розробка створенням UI-системи, яка попередньо включала налаштування конфігурації Vite та створення бібліотеки компонентів. Конфігурація дозволила значно пришвидшити побудову сторінок та покращити якість реалізації функціональних частин. RCL містить всі статичні вебресурси, компоненти, сервіси, необхідні для обох реалізацій: сайту та кросплатформного застосунку.

Було проведене вичерпне тестування як внутрішніх API, так і в цілому всієї платформи. Найважливішим етапом тут були перевірки на валідність виконання всіх обрахунків, доступ до даних користувачів та навантаження на побудований програмний комплекс. Усі розбіжності та недоліки в роботі програмного продукту були вирішені одразу після їхнього виявлення.

Платформа Azure дозволила швидко та повністю розгорнути середовища для всієї системи NumCalc, з використанням Docker-контейнеризації. Налаштування CI/CD-пайплайну автоматизувало оновлення середовищ при попаданні кожної зміни в основну гілку проєкту в Git-репозиторії.

Побудований програмний комплекс може бути вдосконалений шляхом додавання нових чисельних методів та можливих математичних задач. Також допускається вдосконалення відображення покрокових операцій в алгоритмі задля покращення UX. Користувачі реалізованої платформи можуть долучатися до розвитку проєкту: від розробницької частини, оскільки проєкт є відкритим та публічним, до надання власного відгуку про свій досвід користування. Поточна реалізація проєкту дозволяє користувачам виконувати математичні обчислення з використанням чисельних методів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Видрик Ю. С. Застосунок на Blazor для побудови графіків та візуалізації математичних обчислень. Концепт науки XXI: стратегії, методи та наукові інструменти: матеріали X Міжнародної студентської наукової конференції, м. Рівне, 8 травня, 2026 рік. ГО «Молодіжна наукова ліга». Вінниця: ТОВ «УКРЛОГОС Груп», 2026. С. 197-199.

2. He J.-H. An old babylonian algorithm and its modern applications. Symmetry. 2024. Т. 16, № 11. С. 1467.

3. Hollings C. D., Parkinson R. B. Ancient Egyptian mathematics in the early 20th century: a mathematical view from Kiel, 1926. British journal for the history of mathematics. 2024. P. 1-54.

4. GeeksforGeeks. Real-Life applications of numerical analysis. GeeksforGeeks. URL: https://www.geeksforgeeks.org/maths/real-life-applications-of-numerical-analysis/ (date of access: 16.02.2026).

5. Литвинов А. Л. Чисельні методи: теорія і практика : навч. посіб. Харків : Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова, 2022. 166 с.

6. Photomath – the ultimate math help app and math explained. Photomath. URL: https://photomath.com/ (date of access: 08.03.2026).

7. Wolfram | Alpha: making the world’s knowledge computable. Wolfram|Alpha: Computational Intelligence. URL: https://www.wolframalpha.com/ (date of access: 09.03.2026).

8. Site for students majoring in computer science. mathros. URL: https://www.mathros.net.ua/en/ (date of access: 09.03.2026).

9. R. Jangassiyev et al. Comparative analysis of cross-platform development methodologies: a comprehensive study. TELKOMNIKA. Telecommunication Computing Electronics and Control. 2024. Vol. 23, no. 1. P. 108-118.

10. Ponggawa V. V. et al. Comparative study of C++ and C# programming languages / Jurnal Syntax Admiration. Vol. 5, no. 12, 2024. P. 5743-5748.

11. ASP.NET core, an open-source web development .NET framework.

Microsoft. UR: https://dotnet.microsoft.com/en-us/apps/aspnet (date of access: 24.03.2026).

12. What is SQL Server? – SQL server. Microsoft Learn: Build with answers in reach. URL: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver17 (date of access: 24.03.2026).

13. IronPython: package compatibility. GitHub. URL: https://github.com/IronLanguages/ironpython3/wiki/Package-compatibility (date of access: 27.03.2026).

14. CSnakes. Anthony Shaw – Python Developer & Author. URL: https://tonybaloney.github.io/CSnakes/v1/ (date of access: 27.03.2026).

15. Hållberg J. Evaluating blazor rendering modes: a comparative study of Interactive WebAssembly, Interactive Server, and Interactive Auto Rendering. 2025. 45 p.

16. The C4 model for visualising software architecture. C4 model. URL: https://c4model.com/ (date of access: 13.04.2026).

17. Leader J. J. Numerical analysis and scientific computation. Taylor & Francis Group, 2022. 582 p.

ДОДАТКИ

Додаток А

Сертифікат про участь в міжнародній науково-практичній конференції

Додаток Б

Розміщення тез в межах збірника та їхній зміст

Видрик Юлія Сергіївна, здобувачка вищої освіти факультету комп'ютерних та інформаційних технологій
Луцький національний технічний університет, Україна

Науковий керівник: Здолбіцька Ніна Василівна, доцент, доцент кафедри комп'ютерних наук
Луцький національний технічний університет, Україна

ЗАСТОСУНОК НА BLAZOR ДЛЯ ПОБУДОВИ ГРАФІКІВ ТА ВІЗУАЛІЗАЦІЇ МАТЕМАТИЧНИХ ОБЧИСЛЕНЬ

Математичні обчислення давно відігравали та продовжують відігравати значну роль у розвитку людства. Дані обчислення є основою в таких сферах, як інженерія, IT-сфера, економіка тощо. Наразі існує чимало різновидів таких обрахунків: аналітичні методи, що вимагають точне розв'язання формул, символьні обчислення, які сфокусовані на маніпуляціях над математичними виразами та їхніх символах, тощо. В даній роботі більшу частину уваги буде надано саме чисельним методам — методам, що допомагають розв'язувати задачі математики, які тяжко або неможливо вирішити аналітичними способами. Такі методи базуються на поітераційному виконанню певних кроків з метою віднаходження розв'язку поставленої задачі з дозволеною похибкою. Сама концепція даних алгоритмів є досить древньою, датується ще до нашої ери, проте не була часто використовуваною через складність постійних однотипних обрахунків. Проте в 20-ому столітті такі методи обрахунків набули особливого значення, за рахунок пришвидшенного розвитку електронно-обчислювальних машин (ЕОМ) [1].

Хоча зараз є чимало методів та пропрацьованих алгоритмів, що базуються на чисельних методах, проте наразі є проблема з тим, що якщо науковці, студенти або викладачі захочуть спробувати застосувати в своїх роботах чисельні методи, то їм доведеться додатково для цього завантажувати та налаштовувати середовище для виконання скриптів, що можуть бути написані на таких мовах програмування, як Python або MATLAB. Окрім того, що користувач може потенційно стикнутися з проблемами під час такої підготовки, користувач може стикнутися з некоректністю виконання скрипта, що може цілком розчарувати у використанні чисельних методів. До недоліків також можна віднести часту відсутність візуалізації самого процесу віднаходження розв'язку задач, що може також збивати спантелику [2].

Проаналізувавши всі описані вище недоліки, було вирішено створити кросплатформенну систему для вирішення типових математичних завдань, яка базувалося б чисто на чисельних методах. Така система зуміла б надати усім зацікавленим користувачам, незалежно від їхньої обізнаності в сфері програмування, дізнатися про те, як типові математичні задачі потенційно могли б бути вирішені з використанням саме чисельних методів, а не аналітичних або графічних.

Для зручності така система представлена у вигляді кросплатформенного застосунку, що включає в себе реалізацію десктопного застосунку для ОС Windows та мобільний застосунок для ОС Android. Вебверсія також представлена на випадок,

якщо б користувач не захотів завантажувати щось додаткове на свої пристрої. Для такого підходу вирішено було обрати Blazor Hybrid технологію [3], що якраз дозволяє .NET-спеціалістам в одному місці писати загальну кодову базу для усіх платформ одночасно. Оскільки в .NET [4] немає так багато підтримуваних NuGet-пакетів для таких типових математичних операцій, як пошук похідної, первісної тощо, було вирішено інтегрувати два «світи»: .NET (імплементація UI та API сервісу для взаємодії) та Python (реалізація основного функціоналу чисельних методів) [5]. CSnakes якраз можна вважати бібліотекою, що дозволила легко та майже безболісно поєднати ці дві мови програмування в один цілісний механізм [6].

UI такої системи є інтуїтивним для користувача та дозволяє йому з легкістю вводити всі необхідні дані та аналізувати отримані результати. Інтеграція системи з Gemini API дозволила реалізувати якісне розпізнавання математичних виразів (чи то рукописних, чи то друкованих), що значно покращила UX додатків. Логування кроків відображає користувачам сам процес того, як той, чи інший метод може по-різному функціонувати при вирішенні певних типів задач [7]. Функціонал з експортом результатів вирішення математичних задач, дозволяє швидко та легко споживачам продукту ділитися отриманими даними з іншими зацікавленими людьми.

Поліпшення користувацького досвіду забезпечується за рахунок можливості реєстрації, що надає доступ до різноманітного функціоналу: історія останніх обрахунків, збереження вхідних параметрів, історія останніх завантажень результатів обчислень. Історія обрахунків фіксується на кожному реалізованому компоненті системи до 100 записів на одного користувача (видаляються записи автоматично за принципом черги або самим споживачем). Збереження вхідних параметрів дозволяє здійснити швидкий ввід часто використовуваних початкових даних для обрахунків. Висвітлення останніх завантажених результатів обчислень надає можливість швидко завантажити рапорт знову, не залежачи від проведення перерахунків у застосунку. На випадок небажання реєструватися в системі, будь-яка людина може скористатися даним програмним забезпеченням без якихось обмежень та незручностей.

Отже, система NumCalc — унікальна в своєму роді та дозволяє людям з різних спеціалізацій зрозуміти та власноруч перевірити як чисельні методи можуть по-різному працювати над вирішенням різних типів математичних завдань з урахуванням своїх специфік. Це допоможе краще популяризувати чисельні методи (тим більше з урахуванням поточного прогресу в розвитку ЕОМ), як один з видів математичних обчислень, та допоможе краще зрозуміти їхню функціональність.

Список використаних джерел:

1. Guarnieri M. A Short Account on Numerical Methods from Antiquity to Computer Era. 2025 IEEE History of Electrotechnology Conference (HISTELCON), м. Bonn, Germany, 30 верес. – 2 жовт. 2025 р. 2025. С. 1–6. URL: https://doi.org/10.1109/histelcon64051.2025.11286077 (дата звернення: 20.03.2026).
2. Challenges and Opportunities in Data Visualization Education: A Call to Action / B. Bach та ін. IEEE Transactions on Visualization and Computer Graphics. 2023. С. 1–12. URL: https://doi.org/10.1109/tvcg.2023.3327378 (дата звернення: 01.04.2026).
3. Hållberg J. Evaluating Blazor Rendering Modes : A Comparative Study of Interactive WebAssembly, Interactive Server, and Interactive Auto Rendering. 2025. 45 р. (дата звернення: 26.03.2026).

4. ASP.NET Core Blazor Hybrid. Microsoft Learn: Build with answers in reach. URL: https://learn.microsoft.com/en-us/aspnet/core/blazor/hybrid/?view=aspnetcore-10.0 (дата звернення: 27.03.2026).
5. Multi-Language Software Development: Issues, Challenges, and Solutions / H. Yang та ін. IEEE Transactions on Software Engineering. 2024. С. 1–21. URL: https://doi.org/10.1109/tse.2024.3358258 (дата звернення: 23.03.2026).
6. CSnakes. Anthony Shaw - Python Developer & Author. URL: https://tonybaloney.github.io/CSnakes/v1/ (дата звернення: 26.03.2026).
7. Farooqi M. T. K., Siddique Z., Awan S. M. The effectiveness of educational software and applications for primary students' learning outcomes: A systematic review. Journal of Excellence in Social Sciences. 2024. Т. 3, № 4. URL: https://doi.org/10.69565/jess.v3i4.347 (дата звернення: 01.04.2026).

Додаток В

Архітектурні діаграми

Рисунок В.1 – Архітектура спроєктованої системи (контекстний рівень)

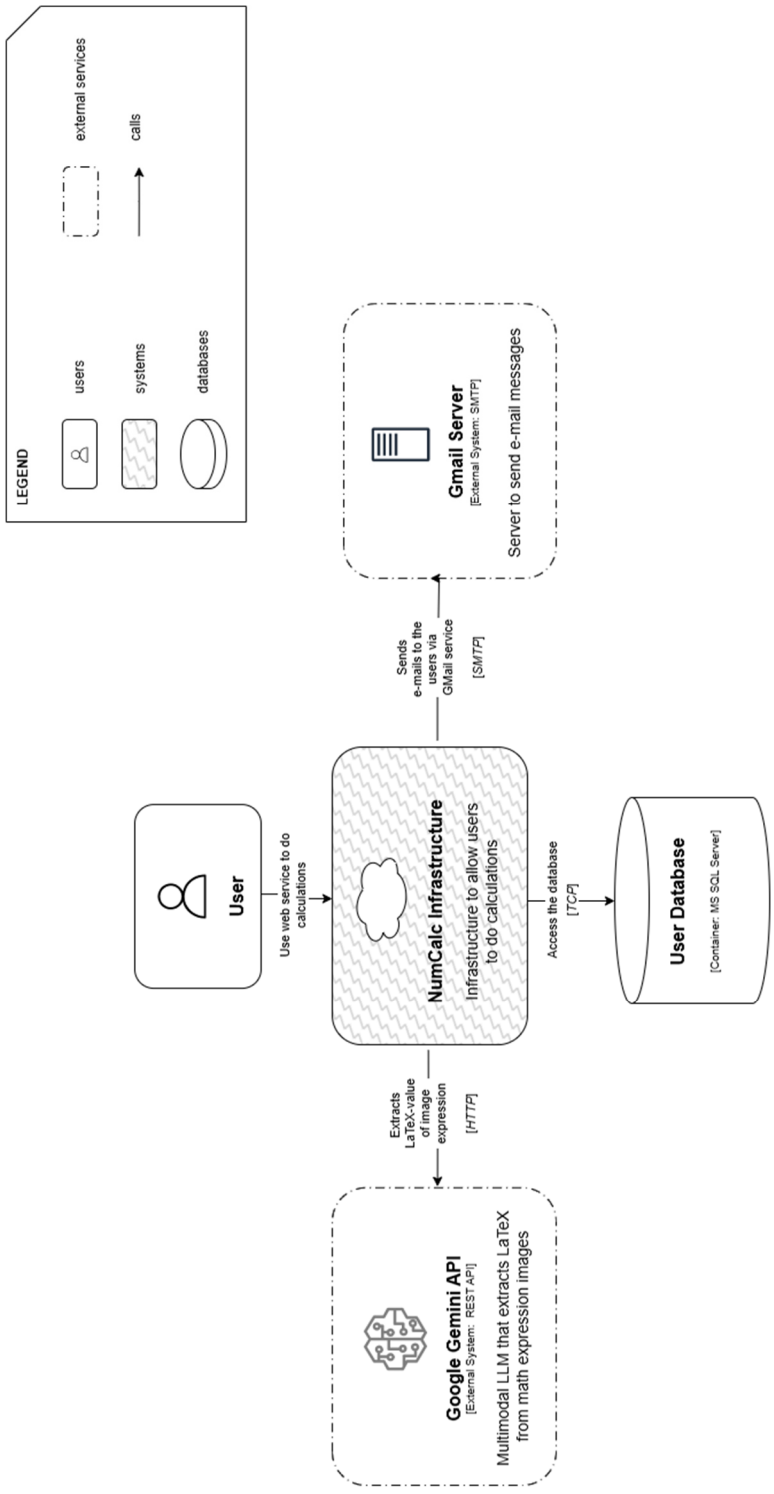


Рисунок В.2 – Архітектура спроєктованої системи (контейнерний рівень)

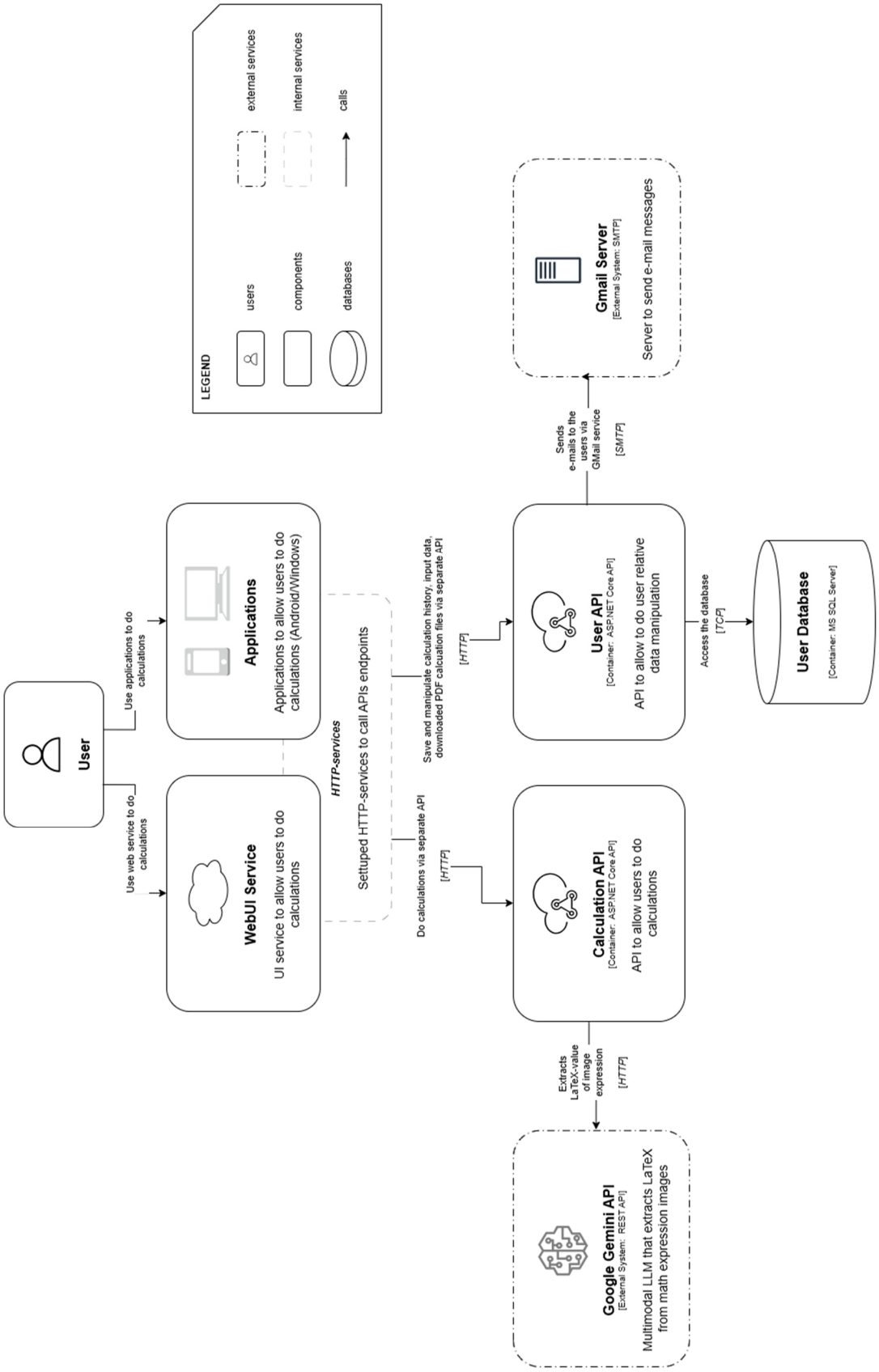


Рисунок В.3 – Архітектура сервісу обрахунків

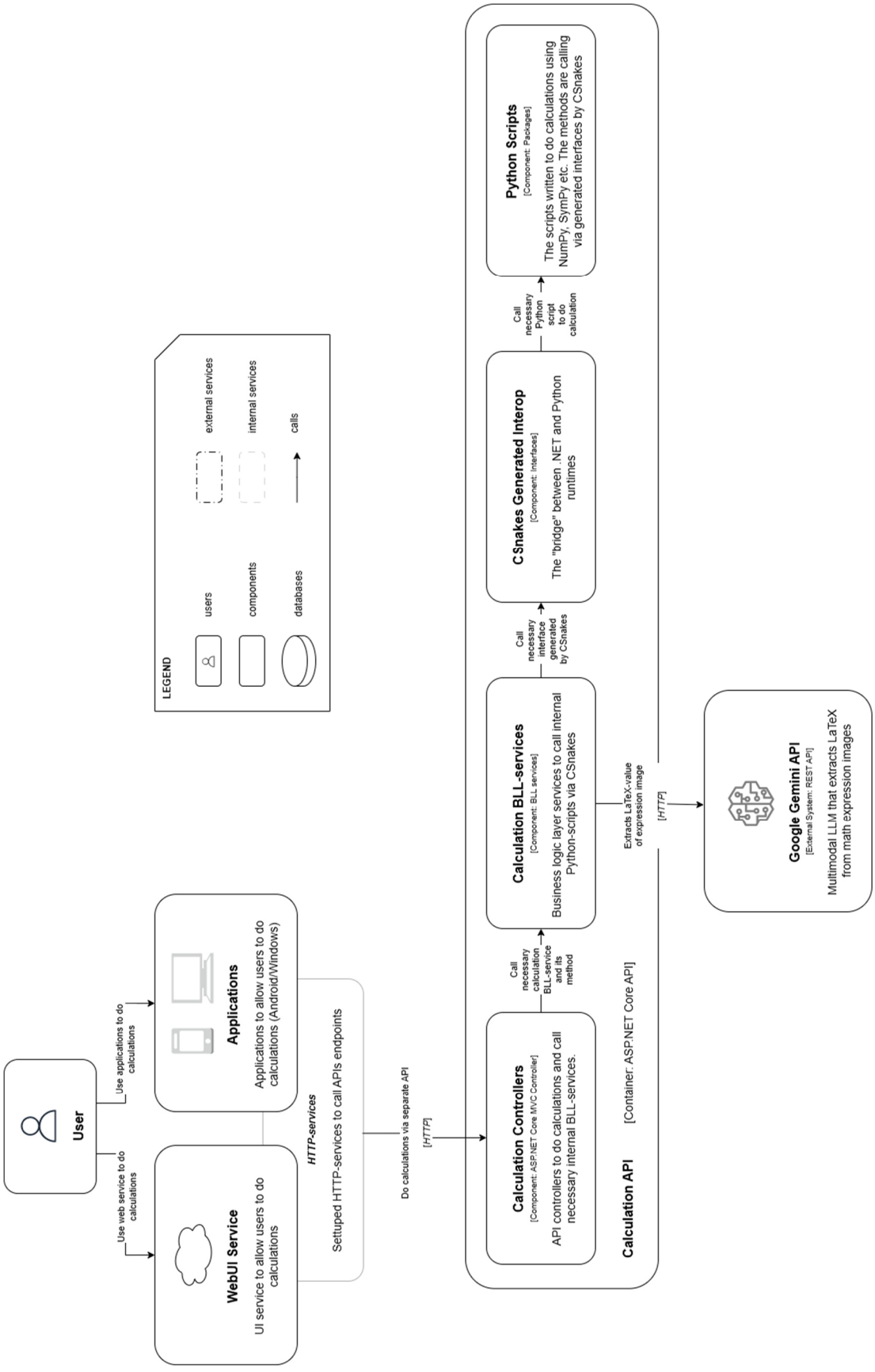


Рисунок В.4 – Опис процесу запиту на пошук вирішення математичної задачі

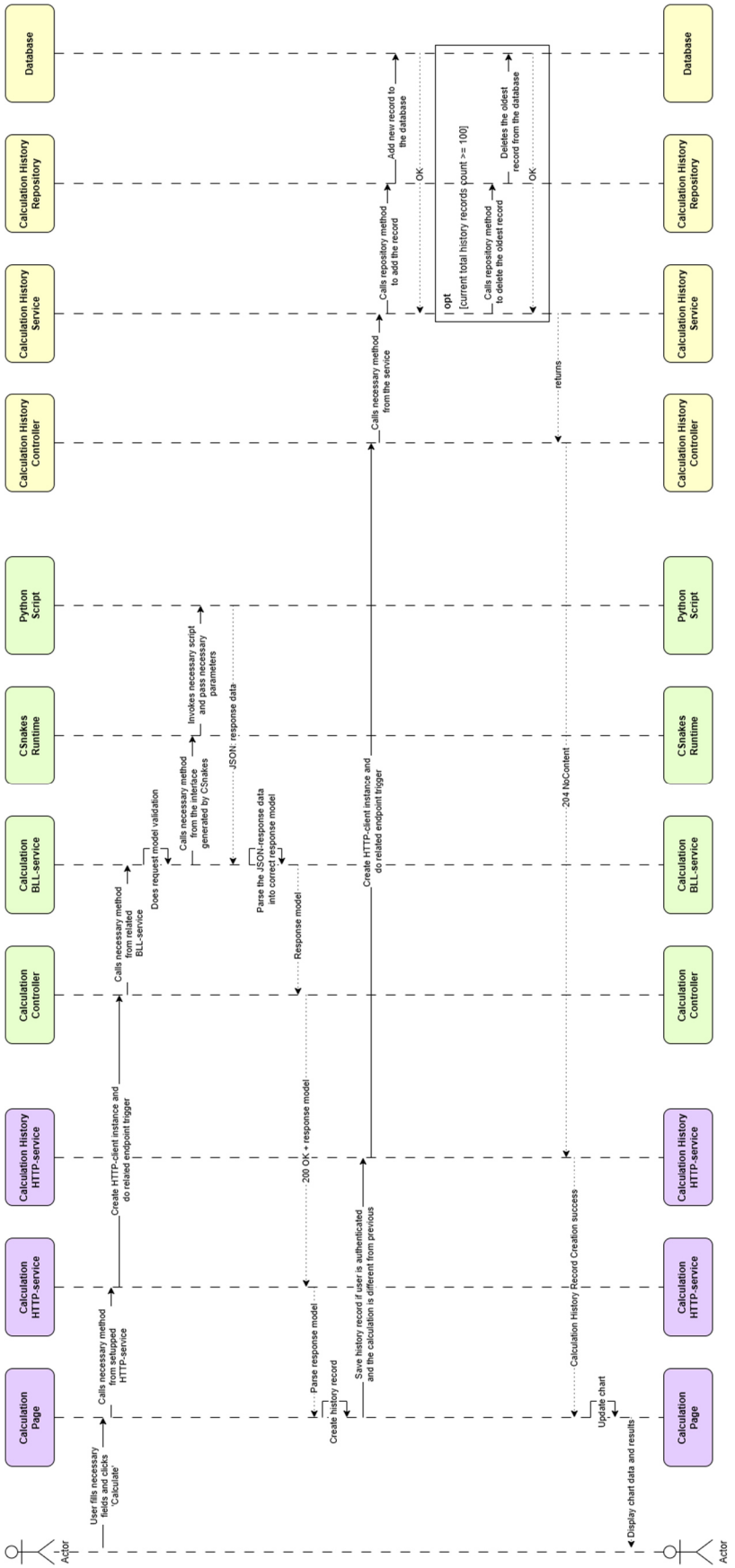
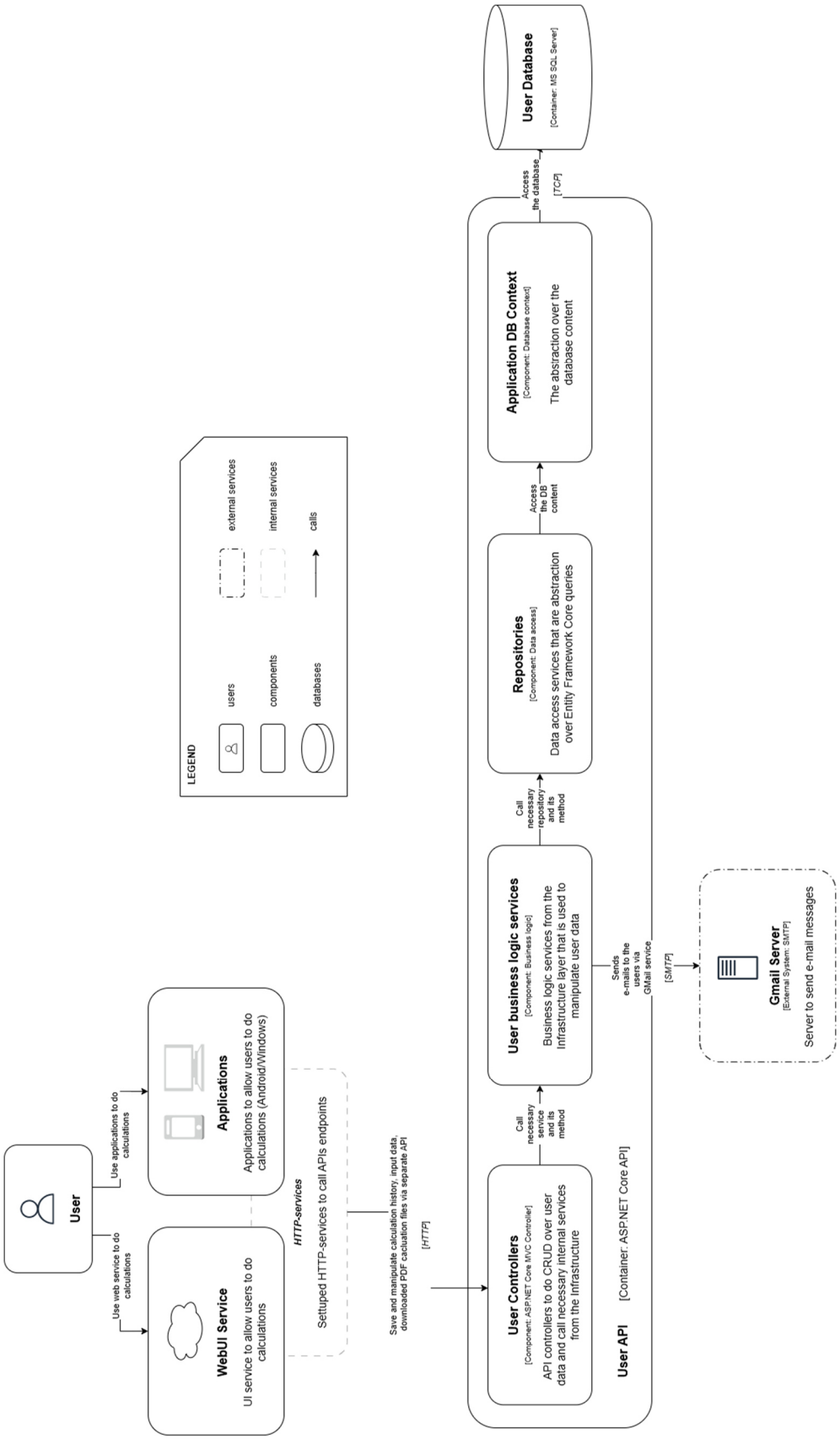


Рисунок В.5 – Архітектура користувацького API



Додаток Г

Повні лістинги коду

Лістинг Г.1 – Структура допоміжного класу PythonResponseUtils

```
public static class PythonResponseUtils
{
    private static readonly JsonSerializerOptions JsonOptions =
new()
    {
        PropertyNamingPolicy = JsonNamingPolicy.SnakeCaseLower,
        PropertyNameCaseInsensitive = true
    };
    public static T UnwrapOrThrow<T>(this string jsonEnvelope)
    {
        PythonResponse<T>? envelope;
        try
        {
            envelope =
JsonSerializer.Deserialize<PythonResponse<T>>(jsonEnvelope,
JsonOptions);
        }
        catch (JsonException ex)
        {
            throw new
CustomException(NumCalcErrorCode.InvalidJson, $"Failed to parse
Python response from gotten JSON: {ex.Message}");
        }
        if (envelope is null)
            throw new
CustomException(NumCalcErrorCode.EmptyResponse, "Python response
is null");
        if (envelope.Failure is not null)
        {
            var errorCode =
Enum.TryParse<NumCalcErrorCode>(envelope.Failure.Code, true, out
var code) ? code : NumCalcErrorCode.UnknownError;
            throw new CustomException(errorCode,
envelope.Failure.Message);
        }
        if (envelope.Success is null)
            throw new
CustomException(NumCalcErrorCode.UnknownError, "Response was not
generic failure, but data is missing.");
        return envelope.Success;
    }
}
```

кінець лістингу Г.1

Лістинг Г.2 – Структура GeminiOcrProvider

```
public class GeminiOcrProvider(HttpClient httpClient,
IConfiguration configuration) : IOcrProvider
{
    private const string Endpoint =
"https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.5-flash:generateContent";
    private const string Prompt =
        "Scan this image and output ONLY the raw LaTeX code for
the mathematical formula. " +
        "Do not use markdown blocks. DO NOT wrap the formula in
$ or $$ signs.";

    private readonly string _apiKey =
configuration["OcrSettings:GeminiApiKey"]?.Trim()
            ?? throw new
CustomException(NumCalcErrorCode.InvalidData, "OCR API key is not
configured");

    private static readonly JsonSerializerOptions JsonOptions =
new()
    {
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
        DefaultIgnoreCondition =
JsonIgnoreCondition.WhenWritingNull
    };

    public async Task<string> RecognizeLatexAsync(string
base64Image, string mimeType, CancellationToken ct = default)
    {
        var payload = new GeminiRequest([
            new GeminiContent([
                new GeminiPart(Text: Prompt),
                new GeminiPart(InlineData: new
GeminiInlineData(mimeType, base64Image))
            ])
        ]);

        var response = await
httpClient.PostAsJsonAsync($"{Endpoint}?key={_apiKey}", payload,
JsonOptions, ct);
        response.EnsureSuccessStatusCode();
        var body = await
response.Content.ReadFromJsonAsync<GeminiResponse>(JsonOptions,
ct);

        return
body?.Candidates?.FirstOrDefault()?.Content?.Parts?.FirstOrDefault
()?.Text ?? string.Empty;
    }

    private sealed record GeminiRequest(GeminiContent[] Contents);
    private sealed record GeminiContent(GeminiPart[] Parts);
```

```
    private sealed record GeminiPart(string? Text = null,
GeminiInlineData? InlineData = null);
    private sealed record GeminiInlineData(string MimeType, string
Data);
    private sealed record GeminiResponse(GeminiCandidate[]?
Candidates);
    private sealed record GeminiCandidate(GeminiContent? Content);
}
```

кінець лістингу Г.2