НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет робототехніки та приладобудування
Кафедра комп'ютерно-інтегрованих оптичних та навігаційних систем

До захисту допущено:
Завідувач кафедри
_____ Надія БУРАУ
«___»_____20 __ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерно-інтегровані системи та технології в приладобудуванні»

спеціальності 174 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка»

на тему: «Адміністративний модуль системи управління цифровими словниками»

Виконав:
студент ІІІ курсу, групи ПГ-п31
Кухар С.С. _____
Керівник:
Асистент, к.т.н. Рупіч С. С. _____
Рецензент:
Доцент, к.т.н. Безугла Н.В. _____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Студент _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут
імені Ігоря Сікорського»

Факультет робототехніки та приладобудування

Кафедра Комп'ютерно-інтегрованих оптичних та навігаційних систем
(повна назва)

Рівень вищої освіти – перший (бакалаврський)
Спеціальність 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка
(код і назва)

Освітньо-професійна програма «Комп'ютерно-інтегровані системи та технології в приладобудуванні»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Н.І. Бурау
(підпис) (ініціали, прізвище)
«___»_____2026 р.

ЗАВДАННЯ
на дипломну роботу студенту
Кухару Станіславу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема дипломної роботи: Адміністративний модуль системи управління цифровими словниками

Науковий керівник: Рупіч Сергій Сергійович, к.т.н. _____ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «__» _____ 20__ р. № _____
2. Термін подання студентом роботи «02» червня 2026 р.
3. Вихідні дані до роботи: реалізувати модуль для адміністрування та управління іноземними словниками; розробити функціонал парсингу та відображення бази слів, можливість формувати індивідуальні тематичні словники; розробити архітектуру програмної реалізації; розроблений модуль має забезпечувати функціонал пошуку, фільтрування та експорту.

4. Питання, що мають бути розроблені:

1. Зробити огляд та аналіз актуальності проблеми.
2. Провести огляд існуючих рішень та підходів для вивчення слів.
3. Визначити вимоги до адміністративного модуля.

4. Розробити архітектуру та структуру програмної реалізації, основні модулі та підсистеми.

5. Описати функціональність та особливості системи.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): таблиці, графіки, рисунки тощо.

6. Дата видачі завдання 01.03.2026

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд стану проблеми	10.05.2026	
2.	Огляд існуючих рішень та підходів	15.05.2026	
3.	Визначення вимог. Побудова архітектури та структури проекту	18.05.2026	
4.	Розробка програмного забезпечення	28.05.2026	
5	Опис функціональності та особливостей системи	30.06.2026	
6.	Оформлення пояснювальної записки. Підготовка до захисту	02.06.2026	

Студент _____ Станіслав КУХАР

Керівник дипломної роботи _____ Сергій РУПІЧ

АНОТАЦІЯ

Дипломна робота присвячена розробці адміністративного модуля системи управління цифровими словниками. Ця система вирішила проблему складності керування великими масивами лексики під час вивчення іноземних мов. Програмне рішення оптимізувало процеси створення, редагування, навігації та розширеного пошуку в межах бази цифрових словників.

Модуль забезпечує автоматизацію процесів для масового наповнення словників даними. Для цього було реалізовано механізми імпорту інформації із зовнішніх джерел. Важливою вимогою було створення можливості експорту словників для їхнього подальшого практичного використання. Для захисту інформації та керування доступом у системі було впроваджено надійні механізми автентифікації та авторизації.

Було спроєктовано архітектуру програмного комплексу, яка основана на принципі розподілу на логічно незалежні компоненти. Було здійснено обґрунтований вибір технологій та інструментів, що дозволило зробити систему гнучкою для подальшого масштабування, стійкою до системних збоїв та простою у підтримці.

У ході проєктування системи було визначено структуру даних, якими оперує система. Для цього було проаналізовано предметну область та обрано відповідні інструменти й технології для управління даними.

Під час розробки було обрано та застосовано ефективний підхід до тестування системи, що дозволило перевірити якість продукту та коректність роботи його програмної логіки.

Пояснювальна записка дипломної роботи складається з трьох розділів, має 42 рисунки та 42 посилання на літературні джерела.

ABSTRACT

This thesis is dedicated to the development of an administrative module for a digital dictionary management system. This system addresses the challenge of managing large lexical databases when learning foreign languages. The software solution optimizes the processes of creating, editing, navigating, and performing advanced searches within the digital dictionary database.

The module automates processes for bulk data entry into dictionaries. To this end, mechanisms for importing information from external sources were implemented. An important requirement was the ability to export dictionaries for their subsequent practical use. To protect information and manage access, reliable authentication and authorization mechanisms were implemented in the system.

The software complex architecture was designed based on the principle of division into logically independent components. A well-founded selection of technologies and tools was made, which allowed the system to be flexible for future scaling, resilient to system failures, and easy to maintain.

During the system design phase, the structure of the data handled by the system was defined. To this end, the domain was analyzed, and appropriate tools and technologies for data management were selected.

During development, an effective approach to system testing was selected and applied, which allowed us to verify the quality of the product and the correctness of its software logic.

The abstract of the thesis consists of three sections, contains 42 figures, and includes 42 references.

ЗМІСТ

ПЕРЕЛІК АКРОНІМІВ ТА ТЕРМІНІВ

SRS – Spaced Repetition System

NGSL – New General Service List

ООП – Об’єктно Орієнтоване Програмування

СКБД – Система Керування Базами Даних

БД – База Даних

API – Application Programming Interface

RBAC – Role-Based Access Control

CSV – Comma-separated Values

ВСТУП

Знання іноземних мов є невід’ємною частиною сучасної освіти, професійного розвитку та ефективної комунікації у глобалізованому світі. Володіння іноземною мовою відкриває доступ до міжнародних інформаційних ресурсів, наукових досліджень, професійної літератури та значно розширює можливості для навчання й кар’єрного зростання.

Мова має багато складових, проте однією з основних компонентів є лексика. З ростом рівня володіння іноземною мовою помітно зростає лексичний запас. Значне зростання лексичного запасу зумовлює збільшення обсягу словникових масивів, що безпосередньо призводить до об'єктивної складності навігації та управління цими даними.

Процес ефективного засвоєння іноземної мови неминуче пов'язаний з опрацюванням великих масивів нової інформації. З розвитком освітніх технологій виникла необхідність в оптимізації цього процесу за допомогою інформаційних систем, здатних структурувати дані.

Наразі в мережі є безліч платформ та веб-застосунків для кінцевих користувачів, які дозволяють виконувати зручний пошук слів серед великих об’ємів даних, проводити ряд операцій над словами та використовувати в процесі вивчення мови. Проте кожна з таких систем повинна мати компонент для адміністрування, який дозволяє керувати словами та словниками. Адміністративний модуль є важливою складовою, адже з кількістю слів в базі зростає і складність їх управління.

РОЗДІЛ 1

ОГЛЯД СУЧАСНОГО СТАНУ ПРОБЛЕМАТИКИ

1.1 Складнощі при вивченні лексики іноземних мов

В еру сучасних технологій спостерігається значне зростання складності запам’ятовування важливої інформації. Це пояснюється надмірним обсягом даних, що сприймається щодня, порівняно з попередніми періодами. Однією з причин цього є стрімкий розвиток технологій, що робить інформацію загальнодоступною та дає змогу практично кожному поширювати її. Особливу роль також відіграє генеративний штучний інтелект, за допомогою якого створюється величезний обсяг контенту, що споживається щоденно. Тобто інформація стала доступнішою, а процес її створення – значно простішим. Відповідно, мозок вимушений опрацьовувати значні обсяги даних, що робить процес запам’ятовування виснажливим [1].

Як наслідок такого когнітивного навантаження, значна частина інформації фільтрується мозком. Успішне подолання цієї проблеми та розробка дієвих стратегій навчання вимагають спирання на фундаментальні закономірності втрати інформації. Базовою науковою моделлю, що ілюструє динаміку цього процесу, є концепція, що лягла в основу багатьох сучасних освітніх технологій.

Крива забування (англ. *Forgetting curve*) – це графічне представлення того, як з плином часу відбувається забування інформації, що не підлягає повторенню, а також вплив повторень на ймовірність збереження інформації у пам’яті. Це не лінійна крива, на яку впливають такі фактори, як кількість та частота повторень, метод вивчення, наявність емоційного зв’язку та інші [2, 3]. Ідею кривої забування зображено на рис. 1.

Рис. 1. Візуалізація кривої забування [3]

Процес опанування більшості іноземних мов пов’язаний з необхідністю засвоєння значного обсягу лексичних одиниць. Когнітивна складність цього процесу зумовлена відсутністю асоціацій із новими словами, а також високою концентрацією омонімів та паронімів, що створює додаткові перешкоди для ідентифікації слів та їх запам’ятовування.

Кількість слів відповідно до рівня володіння мовою за CEFR (The Common European Framework of Reference for Languages) [4]:

- A1-A2: 1,000-2,000 слів.
- B1-B2: 3,000-6,000 слів.
- C1-C2: 8,000+ слів.

Зазвичай носії мови мають словниковий запас в діапазоні від 15,000 до 20,000 сімейств слів [5].

Таким чином, під час вивчення мови, зокрема її лексичної складової, йдеться про тисячі слів, починаючи з початкових рівнів, причому словниковий запас носіїв

мови може сягати кількох десятків тисяч одиниць. Такий обсяг інформації потребує систематичного перегляду. Крім того, правильне структурування такої кількості слів є необхідним, оскільки без відповідного програмного рішення навігація стає вкрай складною.

1.2 Методи та підходи вивчення лексичної складової мови

Враховуючи значний обсяг лексичного матеріалу, що вимагає систематичного засвоєння та довготривалого збереження в пам'яті (згідно з даними, наведеними у попередньому підрозділі), виникає об’єктивна потреба у впровадженні науково обґрунтованих та ефективних методів для організації навчального процесу. Такі методи мають забезпечувати раціональне управління великими масивами слів та оптимізацію інтервалів їх повторення.

Використання карток вважається поширеним та ефективним інструментарієм для вивчення іноземної лексики. Універсальний характер цього методу зумовлює можливість його імплементації в різноманітних сферах знань, оскільки варіативність змістового наповнення та форм адаптації карток практично не має концептуальних обмежень [6].

Ефективність процесу засвоєння іноземної лексики визначається не стільки інструментарієм, скільки методологічним підходом, що застосовується в поєднанні з цими засобами. Як було зазначено в попередньому підрозділі, опанування мови на будь-якому рівні передбачає оперування тисячами лексичних одиниць. На додаток до проблеми великого обсягу даних, виникає когнітивна складність, пов'язана з нерівномірністю засвоєння інформації. Ця загальноосвітня проблема характеризується тим, що частина матеріалу запам'ятовується відносно легко, тоді як інша потребує значних зусиль і систематичного повторення. Враховуючи складність процесів, що лежать в основі формування довготривалої пам'яті, існує об'єктивна необхідність у впровадженні науково обґрунтованих, ефективних підходів,

спрямованих на оптимізацію навантаження та підвищення результативності навчального процесу.

Система інтервального повторення (англ. *Spaced Repetition System, SRS*) – це метод для вивчення та повторення нової інформації, який базується на збільшенні інтервалу між повторенням з кожним разом, коли інформацію успішно пригадано [7].

Базові кроки методу інтервального повторення [7]:

1. Вивчили нову інформацію.
2. Повторюємо інформацію перед тим, як забути її (зазвичай короткий проміжок часу, декілька годин).
3. Якщо можемо пригадати інформацію, збільшуємо наступний інтервал повторення.
4. Якщо інформацію пригадати важко, зменшуємо наступний інтервал.

Система Лейтнера (англ. *Leitner System*) – це підхід до вивчення інформації який чудово підходить при вивченні інформації у форматі карток. Підхід представлений та названий на честь його автора Sebastian Leitner. Головна ідея цього підходу, систематично підтримувати організації інформації, що вивчається, розподіляючи її за групами. Зазвичай ці групи визначаються частотою повторення або ж терміном, через який нам слід повернутися до певної інформації [6].

У процесі роботи з великими обсягами лексики виникають щонайменше дві фундаментальні проблеми:

- Складність визначення моменту засвоєння: важко об'єктивно зрозуміти, на якому етапі інформацію можна вважати остаточно вивченою, щоб припинити її часте повторення без ризику втратити з пам'яті.
- Відсутність оптимальних інтервалів: складно самостійно та ефективно визначати терміни, через які потрібно повторити кожну окрему одиницю інформації для її надійного переходу в довгострокову пам'ять.

Система Лейтнера усуває ці недоліки, запроваджуючи чіткий алгоритм сортування карток за рівнем знань, що дозволяє збільшувати інтервали для знайомих слів та частіше повертатися до тих, які викликають труднощі.

Розглянемо принцип роботи системи Лейтнера на детальному прикладі:

Першим кроком, після визначення обсягу інформації, що підлягає вивченню, здійснюється визначення низки груп із різними інтервалами повторення. Приклад груп розподілених за інтервалами повторення наведено на рис. 2.

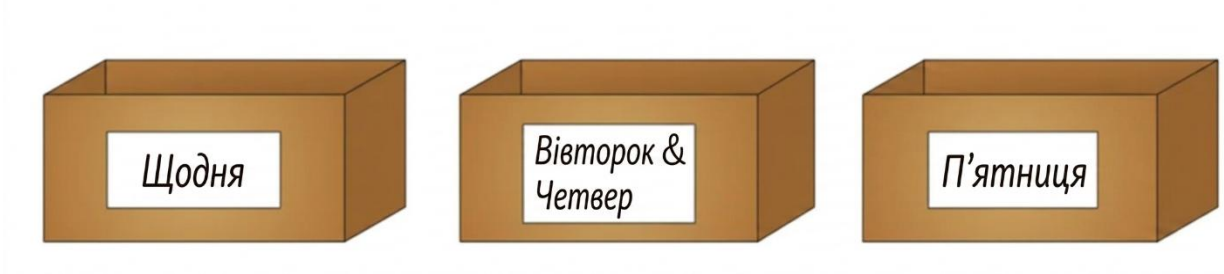

Рис. 2. Розподіл на групи за інтервалом [6]

На початку процесу (перший день) здійснюється перевірка кожної картки. Можливі два результати: успішне пригадування інформації або певні складнощі з пригадуванням.

У разі успішного пригадування інформації на картці, картка переміщується до наступної групи. Випадок ілюстровано на рис. 3.

У випадку, коли пригадування інформації відчувається помітно складним, картка повертається до групи 1. Випадок зображено на рис. 4.

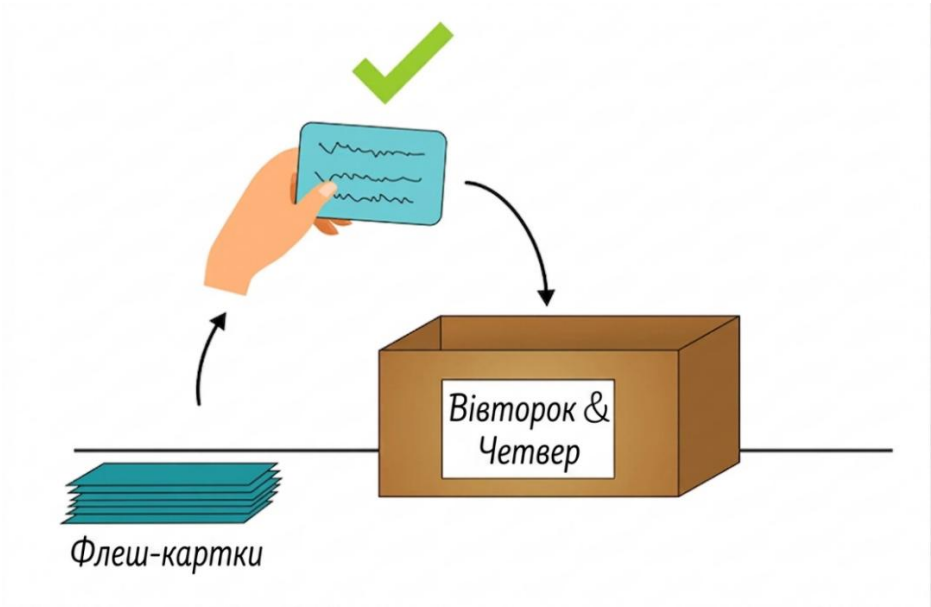

Рис. 3. Успішно пригадали інформації групи 1 [6]

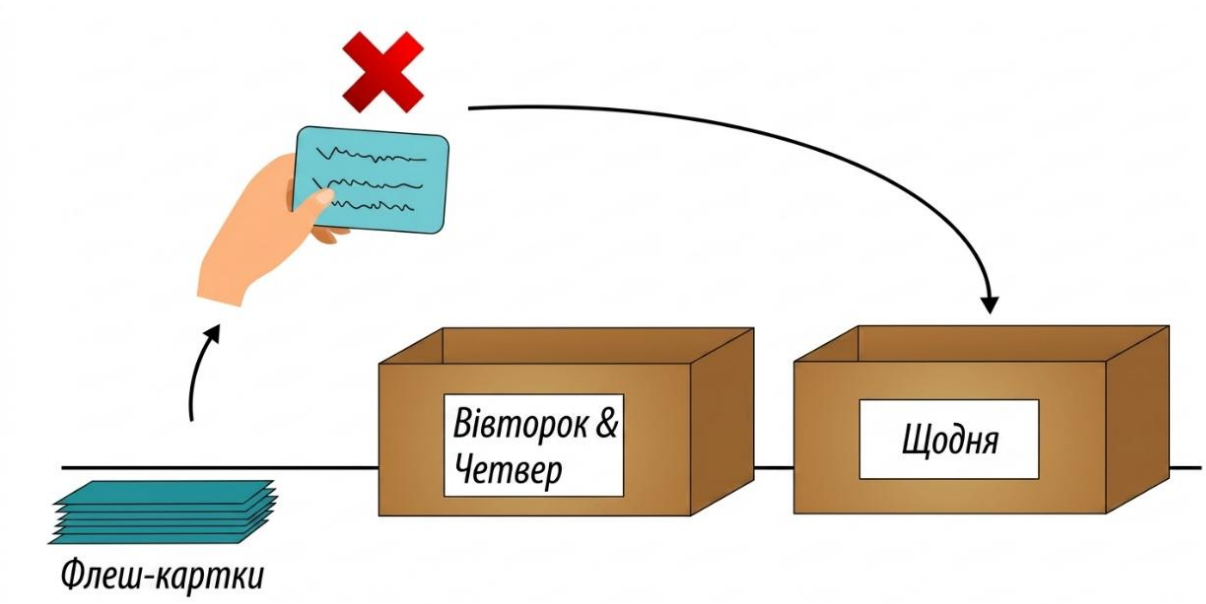

Рис. 4. Випадок коли пригадати інформацію складно [6]

Алгоритм є ідентичним для решти груп. Тобто під час перегляду інформації з будь-якої групи у разі успішного пригадування картка переміщується до наступної групи. Якщо ж пригадування слова викликає труднощі, картка повертається до першої групи. Описаний принцип наведено на рис. 5.

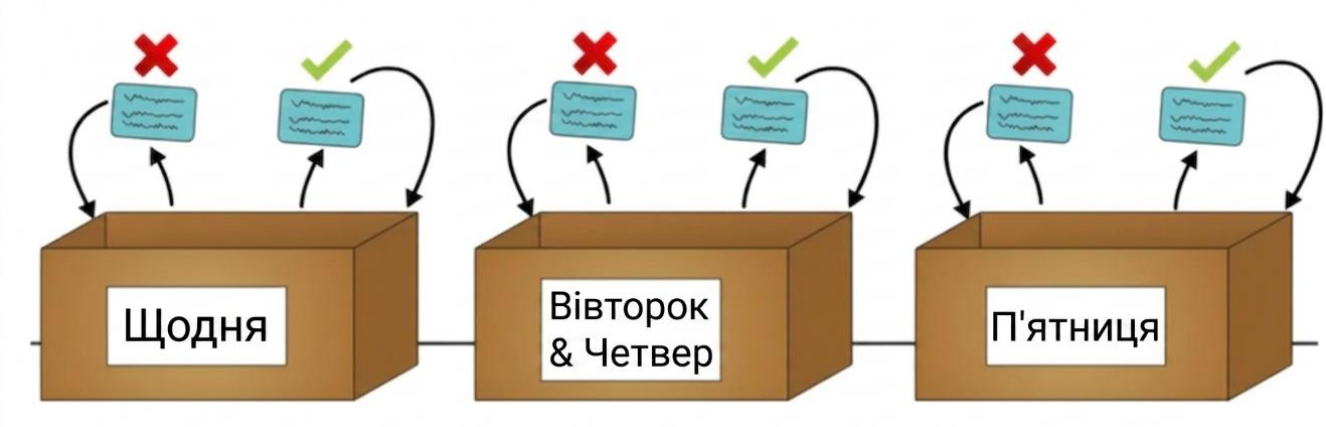

Рис. 5. Ілюстрація принципу роботи системи Лейтнера [6]

Система Лейтнера (Leitner System) визнана не лише ефективним засобом для систематизації навчального матеріалу, але й науково обґрунтованим методом інтервального повторення, що кардинально змінює підходи до засвоєння нової інформації. Традиційні методи засвоєння інформації, як було продемонстровано,

часто супроводжуються такими методологічними недоліками: виникнення хибної ілюзії засвоєння матеріалу в короткостроковій перспективі та нераціональне витрачання часу на повторення вже засвоєної лексики за відсутності ефективних алгоритмів інтервального повторення.

Незважаючи на низку однозначних переваг, які надає метод Лейтнера для оптимізації навчального процесу та розв'язання фундаментальних проблем запам'ятовування, його практична реалізація, особливо в умовах роботи з великими словниковими базами, вимагає значних організаційних зусиль. Тобто виникає об’єктивна необхідність у подальшій автоматизації та вдосконаленні цього процесу.

1.3 Потреба програмного рішення

Незважаючи на високу теоретичну ефективність системи Лейтнера та інтервальних повторень, їх фізична реалізація стикається зі значними практичними перешкодами. Головна проблема полягає в масштабуванні: підхід ідеально працює для кількох десятків карток, але стає некерованим при серйозному вивченні мови.

Цю складність можна розділити на два взаємопов'язані аспекти:

- Велика кількість груп: для досягнення максимальної ефективності алгоритму потрібна гнучка градація знань. Проте у фізичному форматі створення 10, 20 або більше груп призводить до необхідності організовувати відповідну кількість реальних коробок чи секцій. Визначення та контроль розкладу для кожної з них (наприклад, згадати, що сьогодні день перевірки 7-ї та 14-ї коробок) перетворюється на складне логістичне завдання. Вплив кількості груп на ефективність засвоєння інформації та складність управління матеріалом наведено на рис. 6.
- Велика кількість слів: коли словниковий запас розростається до сотень чи тисяч одиниць, ручне сортування карток після кожної навчальної сесії починає займати занадто багато часу. Значна частина зусиль витрачається на адміністрування процесу, ніж на власне запам'ятовування, що є головною ціллю.

Тобто користувач стикається з подвійним бар'єром: йому фізично важко керувати як великими масивами слів, так і складною архітектурою груп з різними інтервалами. Організаційна рутина починає відводити в тінь головну мету, економія часу та його ефективне використання.

Саме ця проблематика робить інтеграцію програмних рішень не просто бажаною, а необхідною. Програмне рішення дозволяє повністю автоматизувати алгоритм. Програмні рішення дозволяють вирішити наступні завдання: розрахунки інтервалів, створення необмеженої кількість віртуальних груп і миттєве сортування значних обсягів лексики у фоновому режимі, спрощуючи процес вивчення.

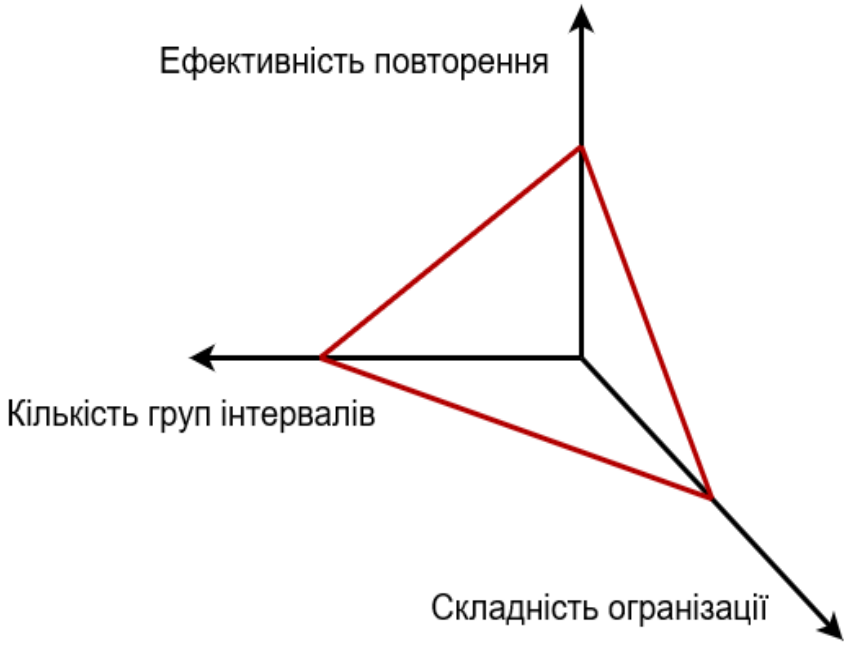

Рис. 6. Ілюстрація впливу кількості груп на ефективність та складність

1.4 Технології та інструменти для програмного забезпечення

У сучасній розробці програмного забезпечення базовим та фундаментальним інструментом є мови програмування. Їх доцільно класифікувати як інструменти

базового рівня, оскільки саме на основі їхнього функціоналу створюються рішення вищого рівня, такі як програмні бібліотеки, фреймворки та платформи [8].

На разі існує безліч різних мов програмування. Значна кількість має свої особливості. Мови програмування можна розглядати поділяючи їх на різні категорії. В даному випадку розглянуто три основні категорії за якими можна класифікувати ці інструменти.

Категорії мов програмування:

- Рівень мови.
- Типізація.
- Підтримувані парадигми.

За рівнем мови класифікують на наступні [9]:

- Мови високого рівня – це мови які відносно легкі в розуміння для людини, адже ключові слова зазвичай використовують звичні слова людської мови.
- Мови низького рівня – це мови наближені до машинного коду, вони складніші в розумінні, проте надають значно більше контролю над системою та апаратним забезпеченням.

Серед розглянутих альтернатив використання мов програмування високого рівня є доцільним у випадках, що не вимагають безпосереднього управління апаратними ресурсами або забезпечення критично високої продуктивності обчислень. Це зумовлено тим, що абстракції високого рівня суттєво оптимізують процес розробки та подальшого супроводу програмного забезпечення.

За типізацією мови по поділяють на наступні [10]:

- Статично типізовані – це мови які передбачають визначення типу одиниці (зазвичай змінної), перед її використанням, зазвичай на етапі оголошення змінної.
- Динамічно типізовані – це мови в яких визначення типу з сторони розробника не обов’язкове. Тип одиниці визначається середовищем в ході виконання.

Дисципліна типізації є ключовим критерієм під час вибору мови програмування. Застосування статичної типізації є особливо доцільним у системах, де сутності предметної області відзначаються високою структурною складністю. У таких випадках система типів виступає механізмом додаткових обмежень, який

формує чіткі архітектурні межі на етапі розробки та підвищує визначеність поведінки програмного забезпечення.

Сучасні мови можуть підтримувати декілька парадигм. Серед основних парадигм можна виділити наступні [11]:

- Імперативна парадигма – є однією з найдавніших концепцій програмування. Її сутність полягає у послідовній зміні стану програми шляхом виконання набору базових інструкцій (операторів).
- Процедурна парадигма базується на принципах імперативної, розширюючи її можливості. Її ключовою особливістю є абстрагування та декомпозиція програмного коду на відокремлені логічні блоки (процедури або підпрограми), що забезпечує можливість їх багаторазового використання та підвищує модульність системи.
- Об’єктно-орієнтована парадигма (ООП) – ця парадигма дозволяє моделювати сутності предметної області у вигляді об’єктів, які функціонують та взаємодіють під час виконання програми. Такий архітектурний підхід суттєво оптимізує процес розробки та проєктування систем у випадках, коли предметна область відзначається високою складністю.

Сучасні адміністративні інформаційні системи об'єктивно потребують інтеграції надійних систем керування базами даних для забезпечення цілісності та безпеки критично важливої інформації. У контексті управління підприємством, бази даних виступають фундаментальним архітектурним компонентом, який дозволяє структурувати великі масиви даних.

База даних (БД) – це певним чином структуровані колекції інформації, які дозволяють зручний доступ, управління та аналіз даних [12].

Система керування базами даних (СКБД) — це система, яка інтегрується в сучасні інформаційні системи для забезпечення цілісності та безпеки критично важливої інформації. Вона дозволяє зручно взаємодіяти з базами даних, які виступають фундаментальним архітектурним компонентом для структурування великих масивів інформації [12].

Structured Query Language (SQL) – це структурована мова запитів для керуванням інформацією в реляційних базах даних [13].

Бази даних досить часто розділяють на дві категорії [12]:

- Реляційні бази даних – це бази даних інформація про сутність в яких представлена у вигляді таблиць, а саме стовбців та рядків. Значення сутностей можуть мати зв’язки. Такі інструменти зазвичай підтримують SQL.
- NoSQL бази даних – це бази даних які розроблені як альтернативи до реляційних баз даних. Вони не обмежені використанням SQL та зазвичай використовуються при роботі з великими обсягами даних неструктурованої інформації.

Застосування реляційних баз даних є доцільним у випадках, коли сутності предметної області характеризуються чіткою, жорстко фіксованою структурою та наявністю складної системи взаємозв'язків. Натомість NoSQL бази даних обирають для обробки масивів даних, які не мають заздалегідь визначеної структури, а також у ситуаціях, коли використання традиційних реляційних моделей призводить до зниження продуктивності системи під час управління великими обсягами інформації [12].

Попри інструментарій який використовується при розробці програмних продуктів, важливим аспектом є середовище виконання. Кожна програма має цільову платформу для якої вона була розроблена. Різниця між типами програмного забезпечення в залежності від цільової платформи наведено на рис. 7.

Типи програмного забезпечення за цільовою платформою виконання [14]:

- Нативні застосунки (англ. *Native applications*) — це програмне забезпечення, розроблене для конкретної апаратно-програмної платформи. До цієї категорії належать застосунки для мобільних пристроїв та персональних комп'ютерів. Окрім апаратної прив'язки, такі програми зазвичай розробляються під певну операційну систему, що дозволяє максимально ефективно використовувати її ресурси.
- Веб застосунки (англ. *Web applications*) — це програмне забезпечення, для якого цільовим середовищем виконання виступає веб браузер. Порівняно з

нативними, такі застосунки здатні функціонувати в будь-якій операційній системі чи на будь-якому пристрої, що підтримує роботу браузера.

- Гібридні застосунки (англ. *Hybrid applications*) — це спеціалізований клас програмного забезпечення, здатний функціонувати на кількох різних платформах. При цьому такі застосунки працюють поза межами браузера забезпечуючи користувацький досвід та функціональність, подібні до нативних застосунків.

Рис. 7. Відмінності типів програмного забезпечення в залежності від цільової платформи

1.5 Існуючі програмні рішення

Reword — сучасний мобільний додаток для вивчення іноземної лексики. Додаток містить значну кількість тематичних словників, що забезпечує структурованість процесу навчання, спрощує навігацію завдяки групуванню слів та дозволяє створювати власні категорії. Інтервали повторень ґрунтуються на принципі кривої Еббінгауза (кривої забування), що сприяє ефективному засвоєнню та тривалому запам’ятовуванню матеріалу. Програма дає змогу створювати, редагувати

та імпортувати картки. Функціонал сервісу передбачає зручне додавання власних слів з автоматичним перекладом та інтегрованою можливістю вибору зображень. Окрім того, додаток має інтуїтивно зрозумілий інтерфейс із корисною статистикою вивчення слів за певний період. Приклади описаного функціоналу зображено на рис.8. [15].

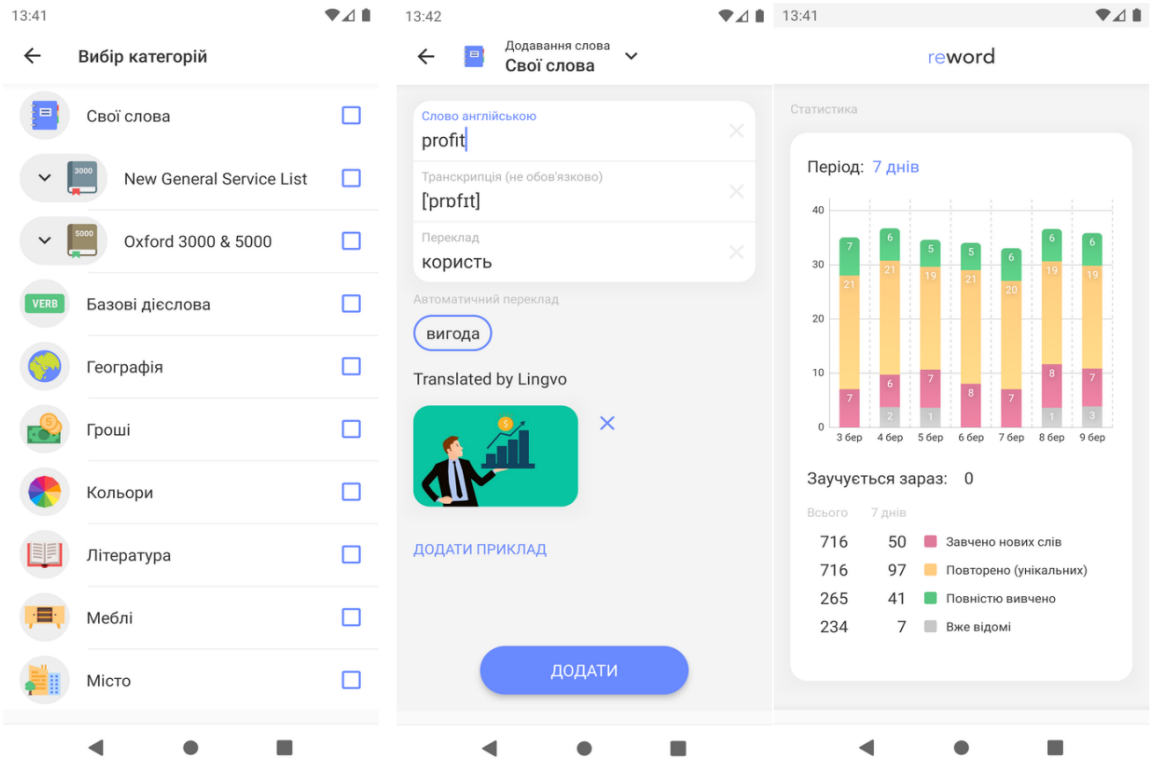

Рис. 8. Функціонал додатку Reword [15]

До переваг Reword можна віднести:

- Доступні словники за темами та інтегровані словники з сторонніх джерел: NGSL та Oxford 3000 & 5000.
- Інтегрований переклад та сервіс пошуку зображень.
- Присутність статистики.
- Можливість імпортувати словники з сторонніх джерел .csv файли, .apkg файли (Anki deck) та експортовані словники з Reword (власні або поширені спільнотою).
- Можливі різні режими повторення слів.

Серед недоліків виділено наступні:

- Слово може потрапити в стан «повністю вивчене».
- Результат перегляду обмежений бінарною відповіддю .

Варто зазначити, що Reword має інтегровані словники готові до використання. Проте він не виключає потребу мануального адміністрування власних словників.

Quizlet - це багатофункціональна освітня платформа, яка базується на використанні інтерактивних карток та пропонує комплексний підхід до засвоєння нових знань. Завдяки колосальній базі готових модулів користувачі можуть вивчати слова за будь-якими темами, групуючи їх у папки для системної навігації. Платформа дозволяє створювати власні набори, редагувати їх та імпортувати дані з таблиць, а вбудовані функції автоматичного перекладу та підбору зображень значно пришвидшують процес підготовки матеріалів. Методологія навчання базується на адаптивних алгоритмах (у преміальній версії), які фокусуються на проблемних зонах користувача, забезпечуючи перехід знань у довгострокову пам'ять через чергування вправ різного типу: від звичайних карток до письмових завдань. Ігровий режим «Підбір» додає елемент змагання, що підвищує залученість, а зручний інтерфейс з детальною статистикою дозволяє наочно відстежувати прогрес за певні періоди. Наявність повної синхронізації між мобільним додатком та веб версією гарантує можливість навчатися будь-де, навіть без доступу до інтернету. Таким чином, Quizlet поєднує в собі гнучкість налаштувань, продуманий інструментарій для створення контенту та сучасні цифрові рішення для ефективного керування власним словниковим запасом і вивчення нових дисциплін [16, 17].

До переваг Quizlet можна віднести:

- Зручний користувацький інтерфейс.
- Наявність різних режимів вивчення слів.
- Можливість використовувати колоди створені спільнотою.
- Інтегрований автоматичний переклад слів.
- Можливість вибирати легко обрати зображення при створенні.

Серед недоліків виділено наступні:

- Значні обмеження в базовій підписці [16, 18].
- Відсутність SRS.

- Обмеження в режимах вивчення.
- Обмеження в кількості навчальних сесій на місяць.
- Обмеження кількості практичних тестів на місяць.

Anki – це багатофункціональна та гнучка платформа для ефективного запам'ятовування інформації, що базується на науково обґрунтованих методах інтервальних повторень та активного відтворення. Основна перевага сервісу полягає в інтелектуальному алгоритмі, який підлаштовується під індивідуальний темп навчання користувача: програма автоматично розраховує оптимальний час для перегляду кожної картки, демонструючи складний матеріал частіше, а засвоєний рідше, що дозволяє мінімізувати зусилля при максимальному результаті. Платформа підтримує мультимедійний контент, дозволяючи інтегрувати в цифрові картки аудіо файли, зображення, відео та навіть складні формули через latex, що робить її незамінною для вивчення іноземних мов. Завдяки відкритому вихідному коду, Anki пропонує безмежні можливості модифікування через тисячі користувацьких плагінів та готові колоди карт, якими ділиться глобальна спільнота. Синхронізація між платформами забезпечує безперервність навчання у будь-якому місці та в будь-який час. Гнучка система тегів, фільтрів та статистичних звітів дозволяє детально аналізувати власний прогрес, перетворюючи хаотичний потік інформації на структуровану довготривалу пам'ять [19].

Серед розглянутих альтернатив Anki має найкращу реалізацію SRS підходу. Слова не мають статусу «Повністю вивчені», натомість інтервал повторення може сягати кількох років. Також при перевірці слів користувач має декілька варіантів відповіді, які впливатимуть на інтервал повторення, що дозволяє значно підвищити ефективність SRS підходу. Приклад варіантів відповіді при перевірці певної інформації наведено на рис. 9.

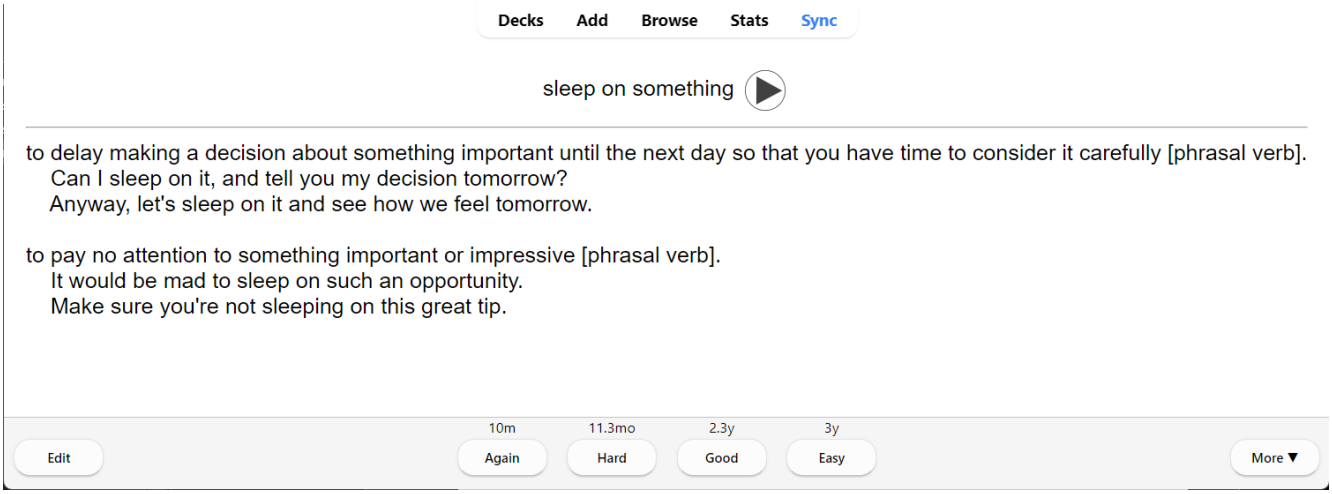

Рис. 9. Варіативність відповідей при перевірці

До переваг Anki можна віднести:

- Комплексна реалізація SRS.
- Синхронізація між платформами.
- Можливість модифікувати картки та їх візуалізацію.
- Достатня кількість метрик та статистики навчання.
- Значна кількість корисних плагінів та можливість створення власних.

Серед недоліків виділено наступні:

- Відсутність безкоштовного додатку на IOS.
- Складність освоєння платформи через широкий спектр можливостей.

1.6 Вимоги до адміністративного модуля управління цифровими словниками

На сучасному етапі розвитку освітніх технологій завдання ефективного запам'ятовування вже має надійні та перевірені програмні реалізації (наприклад, системи інтервального повторення). Проте ключовою проблемою залишається відсутність зручних інтегрованих інструментів для масового управління словниковими базами. Такі популярні платформи, як Quizlet та Anki, фокусуються переважно на процесі відтворення знань і не мають достатньо гнучких вбудованих

засобів для автоматизованого наповнення словників. Вони надають можливість використовувати бази, створені спільнотою, проте такий підхід лише частково маскує проблему. Як тільки у користувача виникає потреба вивчати специфічну, персоналізовану або професійну лексику, він одразу стикається зі значними труднощами, пов’язаними з рутиною ручного створення та організації навчальних карток. Таким чином, фундаментальним недоліком існуючої екосистеми залишається відсутність зручного спеціалізованого середовища для централізованого адміністрування власних словникових баз, їх швидкого наповнення із зовнішніх джерел та подальшого гнучкого експорту в формати, сумісні з популярними платформами вивчення мов.

З огляду на вищезазначене та згідно з технічним завданням на виконання дипломної роботи, базовою метою є реалізація адміністративного модуля системи управління цифровими словниками. Цей модуль покликаний вирішити актуальну проблему централізованого ведення лексичних баз та оптимізувати процеси їх створення і наповнення. На основі аналізу предметної області та технічних специфікацій розроблено наступний комплекс функціональних та архітектурних абстрактних вимог до програмного забезпечення.

Функціональні вимоги:

- Управління даними: система повинна забезпечувати повноцінне управління основними сутностями предметної області: словниками, окремими словами та їхніми значеннями. Передбачається реалізація повного циклу операцій зі створення, читання, оновлення та видалення інформації.
- Автоматизація наповнення баз: модуль має підтримувати механізми швидкого масового додавання лексики. Необхідно розробити функціонал автоматизованого збору та обробки даних із надійних зовнішніх онлайн-ресурсів. Також вимагається реалізація імпорту зі структурованих текстових файлів із забезпеченням гнучкого налаштування відповідності даних до полів внутрішньої бази системи.
- Пошук, фільтрація та агрегація: розроблений модуль має забезпечувати розширений функціонал пошуку та фільтрування лексики. Важливою вимогою є

реалізація наскрізного пошуку (глобального режиму), який здатний об’єднувати однакові слова з різних тематичних словників у єдину логічну сутність для зручного перегляду. Інтерфейс також повинен підтримувати механізми автодоповнення, фільтрації та зручної розбивки результатів на сторінки (пагінації).

- Формування словників та експорт: система повинна надавати можливість гнучкого формування тематичних словників. Окрім того, обов'язковою є наявність механізму експорту сформованих баз у стандартизованих текстових форматах для подальшого їх використання у сторонніх спеціалізованих платформах (наприклад, у системах інтервального повторення).
- Автентифікація та розмежування прав: система потребує інтеграції з сервером управління користувачами. Вимагається чіткий поділ прав доступу на основі ролей: адміністратор, що має повний привілейований доступ до маніпуляцій з даними та автоматизованого імпорту контенту, та звичайний користувач, права якого обмежені лише переглядом та пошуком інформації.

Нефункціональні та архітектурні вимоги:

- Архітектурний підхід: згідно з технічним завданням, необхідно спроєктувати сучасну та масштабовану архітектуру програмної реалізації. Доцільним є застосування розподіленої клієнт-серверної моделі з чітким відокремленням серверної логіки (програмного інтерфейсу) від клієнтського користувацького інтерфейсу, що виконується у веб середовищі.
- Зберігання даних: для надійного збереження складної ієрархічної структури лексичних одиниць система має базуватися на надійній реляційній системі управління базами даних, що гарантує цілісність інформації.
- Безпека та протоколи: захист програмних інтерфейсів та сесій користувачів має здійснюватися із застосуванням сучасних криптографічних стандартів безпеки та індустріальних протоколів авторизації і передачі токенів доступу.

Реалізація вищезазначених вимог дозволить створити гнучкий та ефективний інструмент адміністрування, який повною мірою задовольнить потреби управління навчальною лексикою.

РОЗДІЛ 2

РЕАЛІЗАЦІЯ АДМІНІСТРАТИВНОГО МОДУЛЯ

2.1 Вибір архітектурних підходів

Архітектура програмного забезпечення (англ. *Software Architecture*) – це організована структура компонентів системи та взаємозв'язків між ними, яка визначає шаблон програмного забезпечення в загальних рамках [20].

Архітектура програмного забезпечення відіграє фундаментальну роль у процесі його розробки. Вона безпосередньо впливає на такі критичні параметри та процеси життєвого циклу системи: продуктивність системи, складність інтеграції нових функціональних можливостей та модифікації наявних, рівень складності технічного супроводу програмного продукту, специфіку та обсяг тестування, ефективність моніторингу системних процесів та ідентифікації збоїв, масштабованість та надійність архітектурних рішень.

Багатошарова архітектура (англ. *Layered architecture*) – це один із фундаментальних архітектурних підходів, що передбачає декомпозицію системи на логічні шари (рівні). Кожен шар характеризується чітко визначеними функціями та зоною відповідальності. Згідно з цим підходом, взаємодія компонентів обмежується виключно суміжними рівнями, а ієрархія залежностей між ними має бути строго односпрямованою [21]. Приклад багатошарової архітектури наведено на рис. 10.

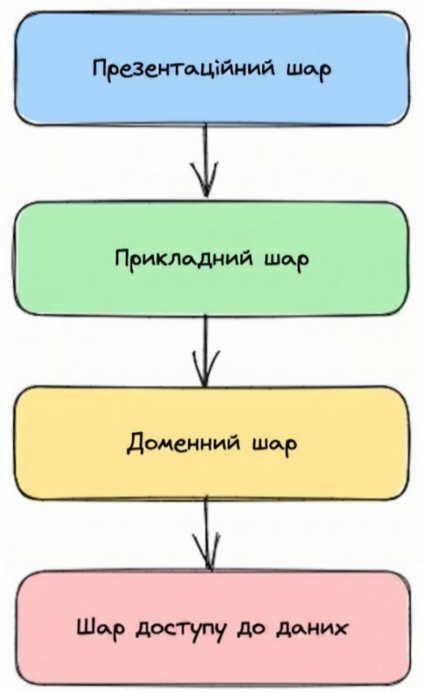

Рис. 10. Візуальне представлення багатошарової архітектури

Чиста архітектура (англ. *Clean architecture*) – це архітектурний підхід до проєктування системи, який закладає розподіл обов'язків між компонентами з метою побудови системи що легко підтримується, масштабується та тестується. Залежності між шарами повинні бути спрямовані всередину [22]. Приклад чистої архітектури наведено на рис. 11.

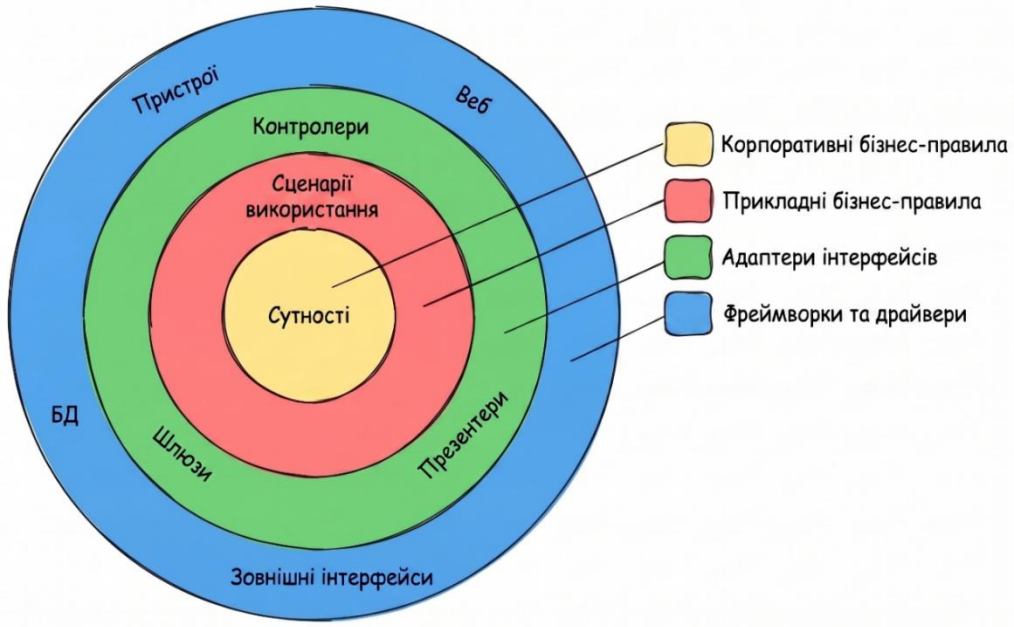

Рис. 11. Візуальне представлення чистої архітектури

Архітектура вертикального зрізу (англ. *Vertical slice architecture*) – це архітектурний підхід, що передбачає декомпозицію системи на автономні зрізи (slices), кожен з яких представляє логіку певної функціональної вимоги. Головною відмінністю цього підходу від традиційних багатошарових архітектурних підходів є групування компонентів за функціями, а не за технічним чи логічним призначенням. Подібна структуризація суттєво спрощує навігацію кодовою базою та супровід проєкту. У межах одного вертикального зрізу можуть поєднуватися елементи різних логічних шарів, проте виключно в обсязі, необхідному для реалізації конкретної функції [23]. Приклад наведено на рис. 12.

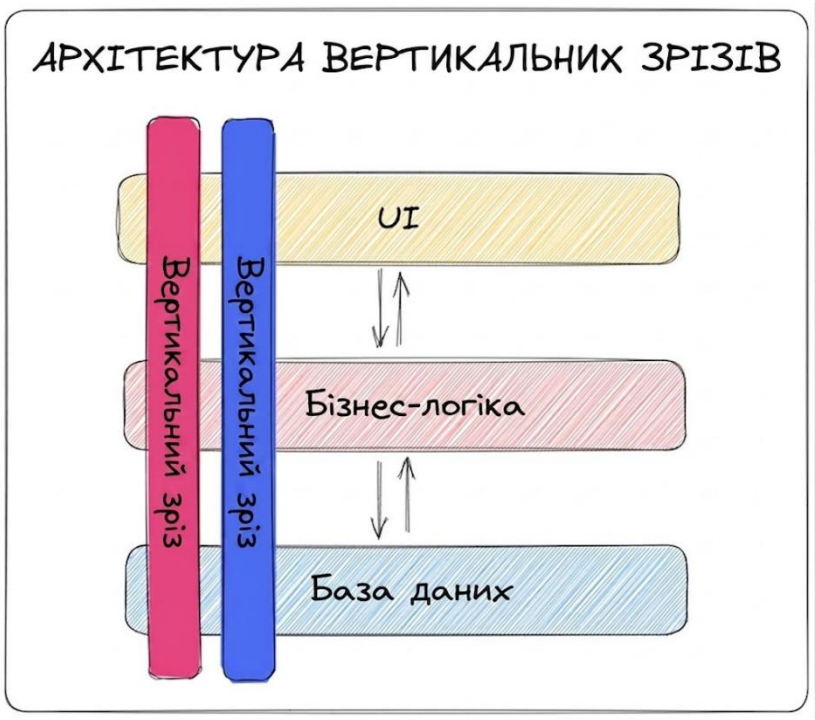

Рис. 12. Візуальне представлення архітектури вертикального зрізу

Застосування підходу чистої архітектури в розробці програмного забезпечення зумовлене необхідністю забезпечення високого рівня незалежності бізнес-логіки від зовнішніх факторів, таких як бази даних, фреймворки чи інтерфейси користувача. Цей підхід значно спрощує процес тестування системи та мінімізує ризики під час подальшого масштабування або заміни окремих технологічних компонентів. Завдяки строгому дотриманню принципу інверсії залежностей, зміни у зовнішніх рівнях не

впливають на внутрішні правила предметної області, що значно підвищує загальну надійність та життєздатність проєкту.

Враховуючи особливості системи та вище згадані архітектурні підходи. За основу було обрано чисту архітектуру. Адаптацію архітектури в проєкті наведено на рис. 13.

Рис. 13. Адаптація чистої архітектури в проєкті

Адаптація архітектури містить на ступні шари та їх компоненти:

- Рівень ядра є центральним компонентом системи, який повністю незалежний від будь-яких зовнішніх бібліотек чи механізмів збереження даних.
  - Сутності: містять основні об'єкти предметної області (наприклад, словники, слова, значення) та фундаментальні правила, що їх описують.
  - Специфікації: використовуються для інкапсуляції бізнес-логіки запитів та критеріїв фільтрації даних, спираючись на абстрактний архітектурний підхід специфікації.

- Рівень сценаріїв використання містить прикладну логіку системи, управляючи взаємодією між сутностями ядра та визначаючи потоки даних для виконання конкретних операцій.
    - Об'єкти передачі даних (DTOs): спеціалізовані об'єкти для передачі даних між рівнями, що запобігають витоку доменних сутностей у зовнішні шари застосунку.
    - Винятки: визначення специфічних класів помилок, що виникають внаслідок порушення правил бізнес-логіки.
    - Інтерфейси: визначають абстрактні контракти для взаємодії з інфраструктурними компонентами, забезпечуючи інверсію залежностей.
    - Сервіси: безпосередньо реалізують прикладні сценарії використання, такі як управління даними, синтаксичний аналіз зовнішніх ресурсів та формування файлів експорту.
    - Валідатори: відповідають за сувору перевірку коректності вхідних даних до моменту їх обробки бізнес-логікою.
- Рівень інфраструктури відповідає за реалізацію технічних деталей та взаємодію із зовнішніми ресурсами, спираючись на абстрактні інтерфейси, визначені на попередніх рівнях.
    - Дані: реалізує механізми доступу до бази даних через архітектурний шаблон репозиторію, включаючи контекст управління даними та конфігурації для взаємодії з реляційною системою управління базами даних.
    - Міграції: забезпечує контроль версіями та автоматизоване управління схемою бази даних за допомогою вбудованих механізмів об'єктно-реляційного відображення.
- Рівень представлення забезпечує взаємодію користувачів та зовнішніх програм із системою, відображаючи інформацію та приймаючи вхідні команди. Він складається з наступних компонентів:
    - API (Application Programing Interface, Серверна точка входу): Відповідає за програмну взаємодію із системою, прийом мережевих запитів,

впровадження залежностей, глобальне перехоплення та обробку винятків, а також маршрутизацію запитів до відповідних сценаріїв використання.

- Клієнт (Клієнтський веб застосунок): Функціонує безпосередньо у браузері кінцевого користувача та відповідає за відображення графічного інтерфейсу, управління локальним станом компонентів і захищену комунікацію з рівнем API через відповідні клієнтські сервіси.

Архітектура системи (англ. *System architecture*) – це архітектурна модель що визначає структуру та поведінку компонентів системи. Компонентами в такій системі можуть бути інші системи, програмні застосунки, сторонні сервіси, мережеві пристрої та апаратне забезпечення [24].

Монолітна архітектура (англ. *Monolithic architecture*) – це методологія проєктування програмного забезпечення, за якої всі компоненти застосунку об'єднуються в єдиний, неподільний блок. У межах цієї архітектури інтерфейс користувача, бізнес-логіка та рівень доступу до даних розробляються, розгортаються та обслуговуються як цілісна, уніфікована система [25].

Архітектура розподілених систем (англ. *Distributed systems architecture*) – це методологія проєктування, за якої система складається з сукупності незалежних програм, що використовують ресурси кількох відокремлених обчислювальних вузлів для досягнення спільної мети. У межах цієї архітектури вузли взаємодіють та синхронізуються через єдину мережу, що має на меті усунення навантажень на конкретні компоненти та центральних точок відмови всієї системи [26].

Мікро сервісна архітектура (англ. *Microservices architecture*) – це підхід до проєктування програмного забезпечення, за якого єдиний застосунок розподілити на сукупність невеликих слабо зв’язаних сервісів, що здійснюють інформаційну взаємодію через мережу. На противагу системам із єдиною, жорстко інтегрованою кодовою базою, кожен такий мікро сервіс діє як повноцінний автономний міні-застосунок, спеціалізований на виконанні конкретної бізнес-функції. Зазначений архітектурний підхід забезпечує високу гнучкість життєвого циклу системи, уможливлюючи повністю незалежний процес розробки, розгортання та масштабування кожного окремого компонента, а також підтримує технологічну

гетерогенність, дозволяючи створювати ці сервіси із застосуванням різноманітних мов програмування та програмних каркасів [27].

В ході виконання було обрано архітектурний підхід розподілених систем зумовлений необхідністю забезпечення високого рівня масштабованості, надійності та гнучкості програмного комплексу. Доцільним є застосування розподіленої клієнт-серверної моделі з чітким відокремленням серверної логіки від клієнтського користувацького інтерфейсу, що виконується у веб середовищі. Такий підхід дозволяє децентралізувати обчислювальні процеси, розділивши систему на незалежні ізольовані вузли що взаємодіють між собою через мережу. Це дозволяє запобігти збоїв що впливатимуть на всю систему, спрощує незалежне масштабування та оновлення окремих модулів без зупинки всієї системи, а також дозволяє ефективно розподіляти ресурси залежно від навантаження. Приклад адаптації даного архітектурного підходу наведено на рис. 14.

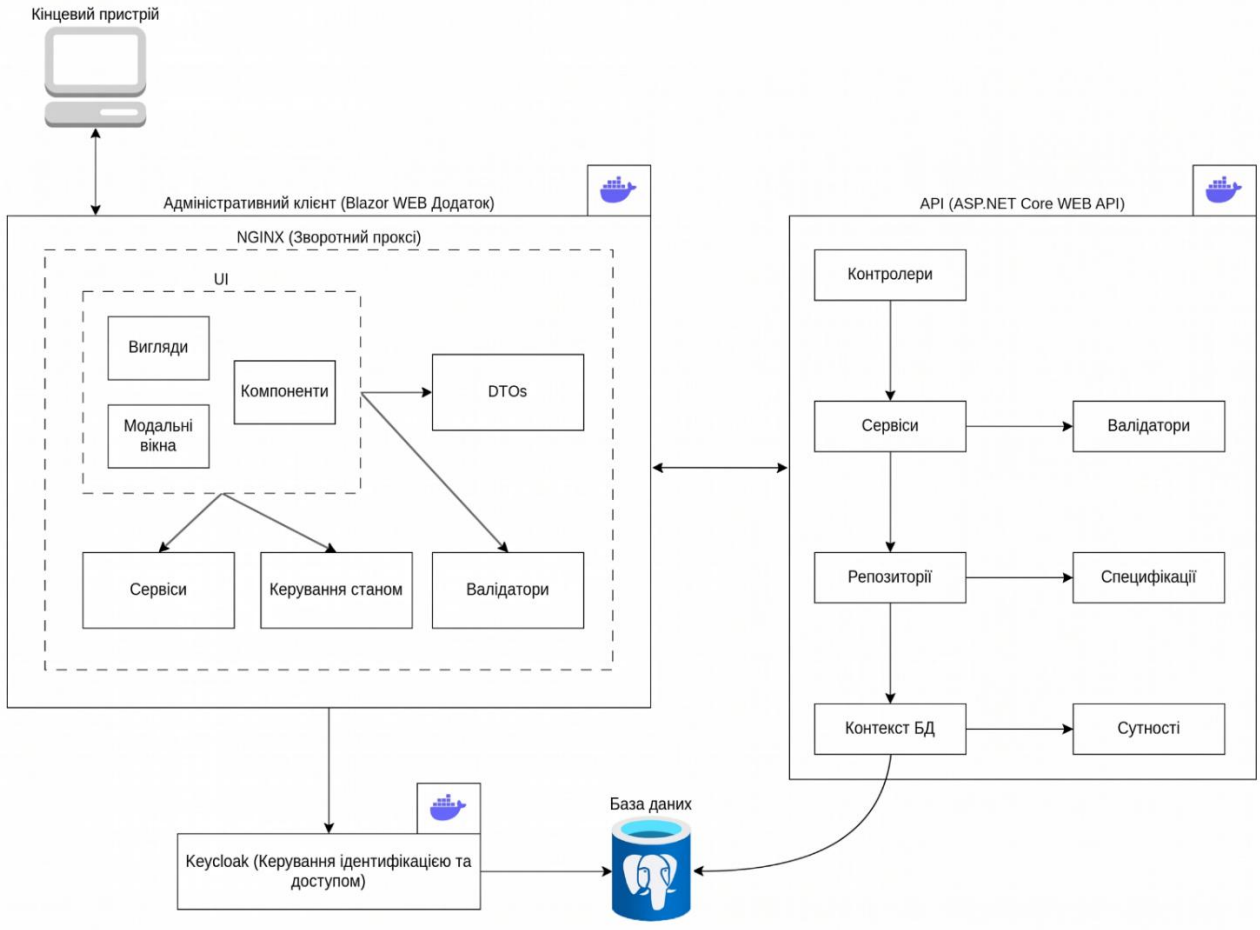

Рис. 14. Адаптація архітектури розподілених систем

Кінцевий пристрій – представляє апаратне забезпечення кінцевого користувача, через яке здійснюється доступ до системи та запуск клієнтського веб середовища.

Адміністративний клієнт – представляє клієнтський веб застосунок, що функціонує безпосередньо у браузері кінцевого користувача. Відповідає за відображення графічного інтерфейсу, управління локальним станом компонентів та захищену комунікацію з рівнем API через відповідні клієнтські сервіси.

Основні компоненти адміністративного клієнта:

- Зворотний проксі-сервер: Відповідає за маршрутизацію вхідних мережевих запитів та ефективну роздачу статичних файлів клієнтського застосунку.
- Користувацький інтерфейс (UI): Включає вигляди, модальні вікна та інтерактивні компоненти.
- Внутрішні модулі: Містить локальні сервіси для виконання запитів, модулі керування станом, валідатори вхідних даних на стороні клієнта та об'єкти передачі даних для формалізації обміну інформацією.

Програмний інтерфейс застосунку (API) – представляє серверну точку входу для програмної взаємодії із системою. Цей компонент відповідає за прийом мережевих запитів, налаштування контейнера впровадження залежностей, глобальне перехоплення та обробку винятків, а також маршрутизацію запитів до відповідних сценаріїв використання. Внутрішній потік даних проходить через наступну послідовність:

- Контролери приймають запити та спрямовують їх далі.
- Сервіси вколючають прикладну бізнес-логіку системи.
- Валідатори здійснюють серверну перевірку коректності вхідних даних.
- Репозиторії та Специфікації абстрагують механізми фільтрації та доступу до даних.
- Контекст БД та Сутності відповідають за об'єктно-реляційне відображення та роботу з доменними моделями предметної області.

Отже, для реалізації адміністративного модуля обрано комплексний підхід, що поєднує принципи чистої та розподіленої архітектури. Застосування підходу чистої архітектури дозволить спростити процес тестування системи та мінімізує ризики під

час подальшого масштабування або заміни окремих технологічних компонентів. Водночас використання розподіленої клієнт-серверної моделі дозволяє децентралізувати обчислювальні процеси, розділивши систему на незалежні ізольовані вузли, що взаємодіють між собою через мережу. Спроєктована з використанням контейнеризації, така архітектура дозволяє зробити програмне рішення гнучким для масштабування та простим у розумінні. У підсумку, інтеграція цих рішень повною мірою задовольняє необхідність забезпечення високого рівня масштабованості, надійності, стійкість до помилок в системі та гнучкості розроблюваного програмного комплексу.

2.2 Вибір технологій та інструментів розробки

Для реалізації програмного комплексу було обрано мову програмування C#. Вибір цієї мови безпосередньо обґрунтовується необхідністю використання строгої статичної типізації, застосування якої є особливо доцільним у системах, де сутності предметної області відзначаються високою структурною складністю. У контексті багаторівневих схем об'єктів розроблюваного модуля система типів виступає механізмом додаткових обмежень, який формує чіткі архітектурні межі на етапі розробки та підвищує визначеність поведінки програмного забезпечення. Крім того, C# повною мірою підтримує об'єктно орієнтовану парадигму (ООП), що дозволяє ефективно моделювати сутності предметної області у вигляді взаємодіючих об’єктів, суттєво оптимізуючи процес розробки та проєктування системи у випадках високої складності домену. Важливою перевагою також є екосистема платформи .NET, яка надає вичерпний інструментарій, сучасні фреймворки та надійні інфраструктурні компоненти, необхідні для швидкої та безпечної розробки масштабованої розподіленої архітектури [28].

ORM (Object-Relational Mapping) – технологія програмування, що забезпечує концептуальне перетворення даних між несумісними системами типів, зокрема між

реляційними базами даних та об'єктно-орієнтованими мовами програмування. ORM дозволяє взаємодіяти з таблицями бази даних безпосередньо через об'єкти та класи, повністю абстрагуючись від ручного написання SQL-запитів [29].

LINQ (Language Integrated Query) – це зручний інструментарій екосистеми .NET, який додає можливості формування запитів безпосередньо до синтаксису мови програмування C#. Він гарантує глибоку інтеграцію функцій роботи з даними з об'єктно-орієнтованою парадигмою мови [30].

Dapper – мікро ORM інструмент для платформи .NET, який відзначається надвисокою продуктивністю. Він забезпечує швидке приведення результатів виконання SQL-запитів до формату представлення у вигляді об’єктів мови C#, проте вимагає від розробника самостійного написання, оптимізації та підтримки SQL-коду [31].

Entity Framework Core (EF Core) – комплексний, повнофункціональний ORM фреймворк для екосистеми .NET. Він гарантує глибоку інтеграцію з об'єктно орієнтованою парадигмою мови програмування (зокрема через інструментарій LINQ), забезпечує автоматичну генерацію SQL-запитів, відстеження змін стану об'єктів та містить вбудований механізм управління міграціями [31].

Використання ORM підходу та фреймворку Entity Framework Core є архітектурно обґрунтованим рішенням для розробки модуля. Вибір повноцінного ORM замість легковагових альтернатив безпосередньо зумовлений високою складністю предметної області. База даних системи містить багаторівневі ієрархічні зв'язки (де словники містять слова, які, у свою чергу, містять відповідні значення). Ручне написання комплексних SQL-запитів для об'єднання таких структур суттєво ускладнило б розробку, збільшило кількість коду та ризик помилок при перетворенні результатів виконання запиту в зручний формат для опрацювання. Натомість EF Core автоматично вирішує ці зв'язки на рівні об'єктних графів [31].

Крім того, EF Core органічно інтегрується в підхід чистої архітектури. Він дозволяє ефективно реалізувати архітектурний шаблон репозиторію та специфікації, надійно ізолюючи логіку доступу до даних на рівні інфраструктури.

ASP.NET Core – це високопродуктивний фреймворк з відкритим вихідним кодом, який призначений для створення сучасних хмарних веб застосунків, серверних систем та програмних інтерфейсів, а також підтримується на багатьох платформах [32].

Для реалізації API в межах розподіленої клієнт-серверної моделі було обрано ASP.NET Core. Це рішення продовжує використання C#, зберігаючи переваги статичної типізації та об'єктно-орієнтованого підходу на сервері. Екосистема .NET надає необхідний інструментарій для побудови масштабованої розподіленої архітектури.

ASP.NET Core органічно інтегрується з підходом чистої архітектури. Вбудований контейнер впровадження залежностей дозволяє ефективно реалізувати їх інверсію між логічними шарами. Це забезпечує надійний прийом мережевих запитів, їхню маршрутизацію та стандартизовану обробку винятків.

Фреймворк також має вбудовані механізми безпеки, що підтримують сучасні протоколи авторизації. Це спрощує інтеграцію з незалежним сервером управління ідентифікацією. Крім того, ASP.NET Core оптимізований для роботи у контейнерних середовищах, що гарантує гнучкість, стійкість до помилок та зручне масштабування системи [32].

Blazor – це сучасний вебфреймворк з відкритим вихідним кодом від компанії Microsoft, який дозволяє створювати інтерактивні клієнтські веб застосунки з використанням мови програмування C# та платформи .NET замість традиційного JavaScript [33].

WebAssembly – це відкритий веб стандарт, який визначає бінарний формат інструкцій для віртуальної машини. Він дозволяє виконувати код, написаний мовами що не підтримуються браузером, без необхідності встановлення додаткових плагінів [34].

Для реалізації клієнтської частини адміністративного модуля було обрано фреймворк Blazor із застосуванням технології WebAssembly. Таке рішення концептуально доповнює вибір ASP.NET Core для серверної частини, формуючи споріднену екосистему на базі платформи .NET. Це дозволяє застосовувати єдину

мову програмування, загальні парадигми та стандартизований інструментарій розробки як на стороні сервера, так і на стороні клієнта.

У порівнянні з традиційними альтернативними рішеннями, що базуються на екосистемі JavaScript, використання Blazor усуває когнітивне навантаження, пов’язане з необхідністю постійного перемикання між різними мовами програмування та архітектурними підходами в межах одного проєкту. Ключовою перевагою є можливість безпосереднього спільного використання кодової бази: об'єктів передачі даних, переліків (Enums) та правил валідації між клієнтським та серверним рівнями. Це суттєво зменшує кількість дубльованого коду та зводить до мінімуму ризик виникнення помилок при серіалізації даних чи невідповідності типів під час мережевої взаємодії [33].

PostgreSQL – це об'єктно-реляційна система керування базами даних з відкритим вихідним кодом, яка вирізняється високою надійністю, масштабованістю та строгою відповідністю сучасним стандартам SQL. СКБД підтримує складні структури даних та надає розширені механізми одночасного доступу [35].

Для організації безпечного та ефективного рівня зберігання даних в адміністративному модулі системи управління цифровими словниками було обрано саме цю реляційну систему. Цей вибір обґрунтовується високою продуктивністю, відповідністю індустріальним стандартам та сумісністю з обраним ORM інструментом – Entity Framework Core.

Застосування реляційної моделі баз даних є цілком доцільним та необхідним архітектурним рішенням для даного проєкту, оскільки сутності предметної області характеризуються чіткою, жорстко фіксованою структурою та складною ієрархічною системою взаємозв'язків. База даних розроблюваної системи містить багаторівневі зв'язки: сутності словників об'єднують колекції слів, які, у свою чергу, містять переліки відповідних значень та їхніх характеристик. Реляційна модель дозволяє ефективно нормалізувати такі дані, мінімізувати їх надмірність та гарантувати цілісність за допомогою зовнішніх ключів [12].

Docker – це відкрита платформа, призначена для автоматизації процесів розгортання, масштабування та управління програмним забезпеченням шляхом його

інкапсуляції у невибагливі, портативні та самодостатні віртуальні середовища – контейнери. Технологія контейнеризації дозволяє об'єднати виконуваний код програми разом з усіма необхідними залежностями, системними бібліотеками та конфігураційними файлами в єдиний стандартизований пакет, що гарантує стабільність та ідентичність виконання застосунку незалежно від характеристик цільової інфраструктури [36].

Для реалізації розроблюваної системи управління цифровими словниками було обрано платформу Docker, оскільки вона органічно доповнює обраний підхід розподіленої архітектури. Кожен архітектурний компонент є ізольованим у власному контейнері зі специфічним, чітко визначеним набором залежностей. Це усуває конфлікти версій програмного забезпечення на рівні первинної операційної системи та підвищує загальний рівень безпеки за рахунок ізоляції процесів [36].

Крім того, застосування даного інструментарію дозволяє використати декларативний підхід до опису інфраструктури. Тобто запуск усього комплексу взаємопов'язаних сервісів, включно з ініціалізацією бази даних та імпортом конфігурацій для сервера авторизації, здійснюється автоматизовано та передбачувано. Такий підхід суттєво спрощує процес локального тестування, робить розроблене рішення гнучким для подальшого незалежного масштабування та підготовленим до інтеграції з сучасними конвеєрами безперервної доставки та розгортання [36].

Сучасні розподілені інформаційні системи об'єктивно потребують інтеграції надійних підсистем управління ідентифікацією та доступом. Застосування таких механізмів є критично важливим для забезпечення загального рівня безпеки програмного продукту, оскільки вони дозволяють централізовано контролювати процеси автентифікації, надійно управляти сесіями та чітко розмежовувати повноваження суб'єктів щодо доступу до системних ресурсів і функціональних можливостей.

Keycloak – це сучасне програмне рішення з відкритим вихідним кодом, яке реалізує концепцію управління ідентифікацією та доступом і забезпечує централізоване керування користувачами та їхніми привілеями. Цей інструмент

функціонує як незалежний сервер авторизації, що бере на себе повну відповідальність за процеси електронної ідентифікації та перевірку повноважень, дозволяючи делегувати ці завдання спеціалізованому компоненту та звільнити основну бізнес-логіку застосунку від необхідності власної реалізації підсистеми безпеки [37].

Головним аргументом на користь Keycloak є його повна архітектурна сумісність із платформою Docker. Наявність оптимізованих образів та підтримка декларативного підходу дозволяють зручно інтегрувати сервер авторизації в інфраструктуру проєкту. Окрім цього, компонент ідеально задовольняє вимогу щодо розмежування прав доступу між адміністраторами та звичайними користувачами. Додатковою перевагою є вбудована підтримка індустріальних стандартів безпеки OpenID Connect, що гарантує надійну взаємодію в межах обраного технологічного стеку [37].

2.3 Проєктування БД

Для організації надійного збереження даних розроблюваного адміністративного модуля застосовується реляційна модель на базі системи керування базами даних PostgreSQL. Застосування реляційної моделі є архітектурно доцільним та необхідним рішенням, оскільки ключові сутності предметної області характеризуються чіткою, жорстко фіксованою структурою та складною ієрархічною системою взаємозв'язків. З метою наочного відображення спроєктованих таблиць, їхніх атрибутів та встановлених між ними нормалізованих зв'язків типу «один-до-багатьох» було сформовано фізичну модель даних. Розроблену схему бази даних наведено на рис. 15.

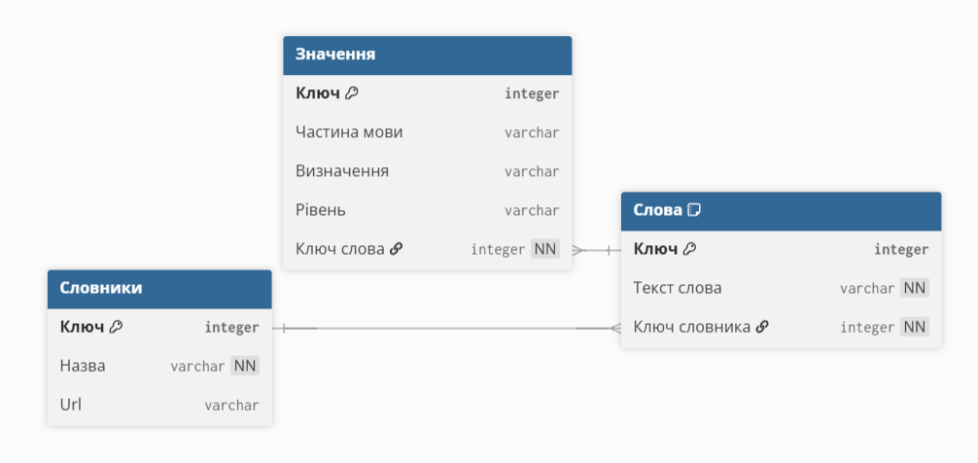

Рис. 15. Схема БД

В ході проєктування було побудовано наступні таблиці:

- Таблиця словників представляє структуру для колекції тематичних словників. Таблиця складається з наступних стовбців:
  - Ключ (integer): первинний ключ, що унікально ідентифікує словник.
  - Назва (varchar, Not Null): обов'язкове поле, що містить назву словника.
  - Url (varchar): не обов’язкове поле для збереження посилання на зовнішнє джерело.
- Таблиця слів представляє окремі лексичні одиниці, які належать до певного словника. Таблиця має наступні стовбці:
  - Ключ (integer): первинний ключ.
  - Текст слова (varchar, Not Null): обов'язкове поле, що містить безпосередньо слово або фразу.
  - Ключ словника (integer, Not Null): зовнішній ключ що вказує на належність слова до конкретного словника.
- Таблиця значень містить детальні визначення та характеристики для кожного конкретного слова. Сутність таблиці має наступні атрибути:
  - Ключ (integer): первинний ключ.
  - Частина мови (varchar): поле, що визначає тип лексеми.
  - Визначення (varchar): текстове пояснення значення слова.

- Рівень (varchar): рівень складності або володіння мовою.
- Ключ слова (integer, Not Null): зовнішній ключ, що прив'язує значення до конкретного слова у таблиці «Слова».

Підхід «Код перший» (англ. *Code first*) – це методологія проєктування баз даних у середовищі EF Core, за якої первинним джерелом істини виступає об'єктно-орієнтований код. Згідно з цим підходом, розробник спочатку створює доменні моделі та описує правила їхнього відображення, після чого фреймворк на основі цих класів автоматично генерує та оновлює фізичну схему реляційної бази даних за допомогою механізму міграцій [38].

Цей підхід дозволяє першочергово спроєктувати незалежні доменні моделі на рівні мови C#, абстрагуючись від адміністрування СКБД за допомогою SQL, та автоматично транслювати складну ієрархію об’єктів у відповідні реляційні таблиці PostgreSQL. Застосування вбудованого механізму міграцій забезпечує супровід контролю версій схеми бази даних за допомогою коду, що дозволяє відстежувати зміни через системи контролю версій [38].

На рис. 16-18 наведено приклади побудови таблиць використовуючи даний підхід в межах EF Core.

```
public class Vocabulary
{
        public int Id { get; init; }
        public string Name { get; set; }
        public string? SourceUrl { get; set; }
        public List<Word> Words { get; set; } = [];
}
public class VocabularyConfiguration : IEntityTypeConfiguration<Vocabulary>
{
     public void Configure(EntityTypeBuilder<Vocabulary> builder)
     {
            builder.HasKey(v => v.Id);
            builder.Property(v => v.Id)
                .HasIdentityOptions(startValue: 1);
            builder.Property(v => v.Name)
                .IsRequired()
                .HasMaxLength(200);
            builder.Property(v => v.SourceUrl)
                .HasMaxLength(2048);
            builder.HasMany(v => v.Words)
                .WithOne(w => w.Vocabulary)
                .HasForeignKey(w => w.VocabularyId)
                .HasPrincipalKey(v => v.Id)
                .OnDelete(DeleteBehavior.Cascade);
     }
}
```

Рис. 16. Код схеми для словників

```
public class Word
{
    public int Id { get; init; }
    public string WordContent { get; set; }
    public int VocabularyId { get; set; }
    public Vocabulary Vocabulary { get; set; }
    public List<Meaning> Meanings { get; set; } = [];
}
public class WordConfiguration : IEntityTypeConfiguration<Word>
{
    public void Configure(EntityTypeBuilder<Word> builder)
    {
        builder.HasKey(w => w.Id);
        builder.Property(w => w.Id)
            .HasIdentityOptions(startValue: 1);
        builder.Property(w => w.WordContent)
            .IsRequired()
            .HasMaxLength(200);
        builder.HasIndex(w => new { w.WordContent, w.VocabularyId })
            .IsUnique();
        builder.Property(w => w.VocabularyId)
            .IsRequired();
        builder.HasMany(w => w.Meanings)
            .WithOne(m => m.Word)
            .HasForeignKey(m => m.WordId)
            .HasPrincipalKey(w => w.Id)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

Рис. 17. Код схеми для слів

```
public class Meaning
{
    public int Id { get; init; }
    public string? LexemeType { get; set; }
    public string? Definition { get; set; }
    public string? Level { get; set; }
    public int WordId { get; set; }
    public Word Word { get; set; }
}

public class MeaningConfiguration : IEntityTypeConfiguration<Meaning>
{
    public void Configure(EntityTypeBuilder<Meaning> builder)
    {
        builder.HasKey(m => m.Id);
        builder.Property(m => m.Id)
            .HasIdentityOptions(startValue: 1);
        builder.Property(m => m.LexemeType)
            .HasMaxLength(100);
        builder.Property(m => m.Definition)
            .HasMaxLength(2000);
        builder.Property(m => m.Level)
            .HasMaxLength(50);
        builder.Property(m => m.WordId)
            .IsRequired();
    }
}
```

Рис. 18. Код схеми для значень слів

Наведені приклади демонструють практичну реалізацію підходу Code First в Entity Framework Core, чітко розділяючи відповідальність між доменними моделями та відповідними класами конфігурацій, що реалізують інтерфейс IEntityTypeConfiguration<T>. Доменні класи містять базові властивості для відображення колонок таблиць та навігаційні властивості для визначення об'єктних зв'язків. Класи конфігурацій забезпечують дотримання принципу єдиної відповідальності, звільняючи моделі даних від інфраструктурних атрибутів і дозволяючи гнучко налаштовувати правила відображення. Зокрема, для кожної сутності визначається первинний ключ та його генерацію за допомогою HasIdentityOptions, а також встановлюються суворі обмеження на обов'язковість заповнення полів IsRequired та їхню максимальну довжину HasMaxLength, що на рівні СКБД трансформується у відповідні типи даних (наприклад, VARCHAR) для оптимізації пам'яті [39].

Особливу увагу в конфігураціях приділено реалізації бізнес-правил та підтримці цілісності даних. Наприклад, для сутності слова створено складений унікальний індекс за полями тексту слова та ідентифікатора словника HasIndex(...).IsUnique(), що на рівні бази даних гарантує відсутність дублікатів одного й того ж слова в межах конкретного словника. Взаємозв'язки типу «один-до-багатьох» між ієрархічними рівнями системи жорстко специфіковані за допомогою методів HasMany та WithOne із вказівкою відповідних зовнішніх та головних ключів. Крім того, застосування налаштування OnDelete(DeleteBehavior.Cascade) ініціює механізм каскадного видалення. А саме знищення словника автоматично призводить до безповоротного видалення всіх підпорядкованих йому слів та пов'язаних із ними значень, що ефективно запобігає появі забутих записів у реляційній базі даних [39].

2.4 Інтеграція підсистеми автентифікації та авторизації

Автентифікація – це процедура електронної ідентифікації, яка полягає у встановленні та логічному або криптографічному підтвердженні справжності заявленого суб'єкта в інформаційній системі. Цей процес здійснюється шляхом перевірки наданих ідентифікаційних даних на їхню ідентичність еталонним значенням, що зберігаються в підсистемі безпеки або на сервері управління користувачами [40].

Авторизація – це процес визначення, надання та контролю прав доступу суб'єкта до певних об'єктів, даних чи функціональних можливостей інформаційної системи. Вона базується на перевірці наявності відповідних повноважень для виконання певних операцій згідно із затвердженою політикою безпеки та обраною моделлю розмежування доступу [40].

Управління доступом на основі ролей (англ. *Role-Based Access Control, RBAC*) – це модель розмежування прав доступу в інформаційних системах, яка базується на призначенні повноважень об’єктам відповідно до їхніх логічних функцій (ролей) у межах організаційної структури, а не на основі індивідуальних ідентифікаторів. У цій архітектурі права на виконання специфічних системних операцій асоціюються виключно з конкретними ролями, після чого користувачам надається авторизація через присвоєння відповідного статусу [41].

Головним обґрунтуванням застосування моделі RBAC є необхідність реалізації ключової функціональної вимоги до модуля — чіткого розмежування прав доступу між двома основними типами користувачів. Цей підхід дозволяє архітектурно розділити повноваження адміністратора, який має привілейований доступ до маніпуляцій з даними та автоматизованого імпорту, і звичайного користувача, чиї права обмежені базовим переглядом та пошуком лексики. Вторинною, але вагомою перевагою є те, що використання рольової моделі у поєднанні з компонентом Keycloak усуває потребу в детальному управлінні окремих облікових записів та

забезпечує централізований захист лексичного контенту від несанкціонованих модифікацій.

Для забезпечення надійного захисту серверної частини розробленої системи на базі ASP.NET Core Web API та інтеграції з сервером управління ідентифікацією Keycloak через механізм JSON Web Token було реалізовано конфігурацію проміжного програмного забезпечення. На рис. 19 наведено фрагмент коду що ілюструє процес налаштування параметрів автентифікації та адаптацію специфічної структури токена Keycloak до стандартів безпеки платформи .NET.

```
...
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.Authority = keycloakAuthority;
        options.RequireHttpsMetadata = false;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = keycloakValidIssuer ?? keycloakAuthority,
            ValidateAudience = false,
            ValidateLifetime = true
        };
        options.Events = new JwtBearerEvents
        {
            OnTokenValidated = ctx =>
            {
                var realmAccess = ctx.Principal?.FindFirst("realm_access")?.Value;
                if (realmAccess is not null)
                {
                    using var doc = JsonDocument.Parse(realmAccess);
                    if (doc.RootElement.TryGetProperty("roles", out var roles))
                    {
                        var identity = (ClaimsIdentity)ctx.Principal!.Identity!;
                        foreach (var role in roles.EnumerateArray())
                            identity.AddClaim(new Claim(ClaimTypes.Role, role.GetString()!));
                    }
                }
                return Task.CompletedTask;
            }
        };
    });
builder.Services.AddAuthorization();
...
```

Рис. 19. Конфігурації автентифікації та авторизації на сервері

Наведений фрагмент конфігурації безпосередньо визначає набір параметрів токена доступу, які підлягають суворій перевірці на стороні серверу. Зокрема, він регламентує процес перевірки автентичності вхідного токена, включаючи перевірку його терміну дії, а також встановлює алгоритм динамічного формування користувацьких ролей на основі вилучених із нього даних. Це забезпечує коректну трансформацію зовнішньої рольової моделі у зрозумілий для системи формат, що є

необхідним етапом для подальшого прийняття рішень щодо авторизації та розмежування доступу. На рис. 20 на ведено приклад використання ролей на стороні серверу для прийняття рішень в межах процесу авторизації.

```
[HttpDelete("{id:int}")]
[Authorize(Roles = "Administrator")]
public async Task<IActionResult> DeleteWord([FromRoute] int id)
{
    await _wordStoreManager.DeleteWordById(id);
    return Ok($"Word by id: {id} removed successfuly.");
}
```

Рис. 20. Приклад використання авторизації в кінцевій точці серверу

Атрибут Authorize дозволяє перевіряти наявність ролей що були сформовані н етапі перевірки токену. Таким чином доступ до даної кінцевої точки буде дозволено тільки користувачам роді «Адміністратор».

Для забезпечення безпечного доступу клієнтського застосунку на базі Blazor WebAssembly до захищених ресурсів системи було реалізовано механізм автентифікації на основі протоколу OpenID Connect. На рис. 21 наведено фрагмент конфігурації, що ілюструє процес підключення клієнтського модуля до сервера ідентифікації Keycloak та налаштування правил обробки сесій і запитів на стороні клієнта.

```
...
builder.Services.AddOidcAuthentication(options =>
{
    options.ProviderOptions.Authority = builder.Configuration["Keycloak:Authority"];
    options.ProviderOptions.ClientId = builder.Configuration["Keycloak:ClientId"];
    options.ProviderOptions.ResponseType = "code";
}).AddAccountClaimsPrincipalFactory<KeycloakAccountClaimsPrincipalFactory>();

builder.Services.AddScoped<ApiAuthorizationMessageHandler>();
...
```

Рис. 21. Конфігурація Keycloak на стороні клієнта

На рис 22. наведено фрагмент, що використовує авторизацію на стороні клієнту. Елемент користувацького інтерфейсу працює в схожий спосіб як атрибути кінцевих точок на серверній частині. Головна відмінність в тому, що він визначає чи потрібно відображати вкладену частину користувацького інтерфейсу користувачу на базі його ролей.

```
<div class="border border-1 rounded m-1 p-2">
    <div class="d-flex justify-content-between align-items-center">
        <div>
            <NavLink href="@($"/word/{Word.Id}")"
                class="fw-bold text-primary text-decoration-none">
                @Word.WordContent
            </NavLink>
        </div>
        <div class="d-flex">
            <AuthorizeView Roles="Administrator">
                <NavLink href="@($"/word/{Word.Id}")"
                    class="btn btn-link btn-sm text-secondary text-decoration-none m-lg-1">
                    @SharedLoc["Label_Edit"]
                </NavLink>
                <Button Class="m-lg-1"
                    @onclick="DeleteWord"
                    Type="ButtonType.Link"
                    Color="ButtonColor.Danger"
                    Size="Size.Small">
                    @SharedLoc["Btn_Delete"]
                </Button>
            </AuthorizeView>
        </div>
    </div>
</div>
```

Рис 22. Використання авторизації на стороні клієнту

2.5 Опис функціональності та особливостей системи

Для оптимізації процесу наповнення лексичних баз в адміністративному модулі реалізовано автоматизовані механізми масового додавання лексики. Відповідно до визначених функціональних вимог, система підтримує два незалежні підходи до імпортування даних із зовнішніх джерел, що забезпечує гнучкість обробки як онлайн-ресурсів, так і локальних структурованих файлів.

Способи імпортування зовнішніх лексичних баз:

- Автоматизований парсинг з онлайн-ресурсів (Oxford): Реалізовано функціонал збору та обробки даних безпосередньо з веб сторінок словника Oxford за допомогою вказаної URL-адреси. Цей процес обробляється на рівні програмного інтерфейсу за допомогою бібліотеки AngleSharp, яка забезпечує ефективний синтаксичний аналіз веб сторінок.
- Імпорт зі структурованих текстових файлів (англ. *Comma-separated values, CSV*): Забезпечено можливість завантаження даних у форматі CSV із подальшим парсингом та гнучким налаштуванням відповідності колонок імпортованого файлу до полів внутрішньої моделі системи. Опрацьовані лексичні одиниці можуть бути збережені як у новостворений, так і у вже існуючий тематичний словник.

Практичну реалізацію автоматизованого процесу вилучення лексичних даних із зовнішніх інформаційних ресурсів, що діє за принципом послідовної структурної трансформації інформації, наведено на рис. 23. На початковому етапі програмний компонент перевіряє валідність вхідних параметрів та виконує звернення до зовнішнього постачальника даних для завантаження HTML-документа за вказаною URL. Наступним кроком є етап синтаксичного аналізу який полягає в алгоритмічному обході об'єктної моделі документа та жорсткій фільтрації вузлів за допомогою ієрархії CSS-селекторів. Це дозволяє точно локалізувати необхідні HTML-елементи та вилучити з них цільові лексичні одиниці, відсіюючи зайвий контент веб сторінки. Завершальна стадія передбачає асинхронне генерування потоку вилучених текстових даних для їхньої подальшої нормалізації, адаптації до внутрішньої доменної моделі та збереження в системі.

```
public class SimpleOxfordDictionaryParser
    : IWordParser<string>
{
    public async IAsyncEnumerable<string> GetWordListByLinkAsync(string link)
    {
        if (string.IsNullOrEmpty(link))
        {
            throw new ArgumentNullException("Passed url can't be neither empty nor null.");
        }

        IConfiguration config = Configuration.Default.WithDefaultLoader();
        IBrowsingContext context = BrowsingContext.New(config);
        IDocument document = await context.OpenAsync(link);

        if (document == null)
        {
            throw new NullReferenceException("The content of source shouldn't be a null value.");
        }

        IHtmlCollection<IElement>? wordsAsLiElements = document
            .QuerySelector("div#ox-container")?
            .QuerySelector("div.responsive_container.xonecolumn")?
            .QuerySelector("div.responsive_row")?
            .QuerySelector("div.responsive_entry_center")?
            .QuerySelector("div.responsive_row.responsive_entry_center_wrap")?
            .QuerySelector("div#main_column")?
            .QuerySelector("div#informational-content")?
            .QuerySelector("div#wordlistsContentPanel")?
            .QuerySelector("ul.top-g")?
            .QuerySelectorAll("li");

        if (wordsAsLiElements == null)
        {
            throw new TheSourceIsNotAppropriateException("The source isn't appropriate for target purpose.");
        }

        foreach (IElement word in wordsAsLiElements)
        {
            string? wordContent = word.QuerySelector("a")?.TextContent; // tag <a> which contains the text of word

            if (!string.IsNullOrEmpty(wordContent))
            {
                yield return wordContent;
            }
        }
    }
}
```

Рис. 23. Реалізація сервісу для зчитування слів з Oxford словника

Альтернативним механізмом заповнення лексичних баз є процес імпортування структурованих даних із локальних текстових файлів формату CSV, логіку роботи та блок-схему алгоритму якого наведено на рис. 24. На початковому етапі алгоритм приймає масив текстових рядків файлу та символ-розділювач, здійснюючи базову перевірку вхідного потоку для запобігання обробці порожніх файлів або структур без колонок. Процес синтаксичного аналізу розпочинається з виокремлення першого рядка для формування списку заголовків, після чого відбувається ітеративна обробка решти непорожніх рядків із послідовним розбиттям їх на окремі атрибути.

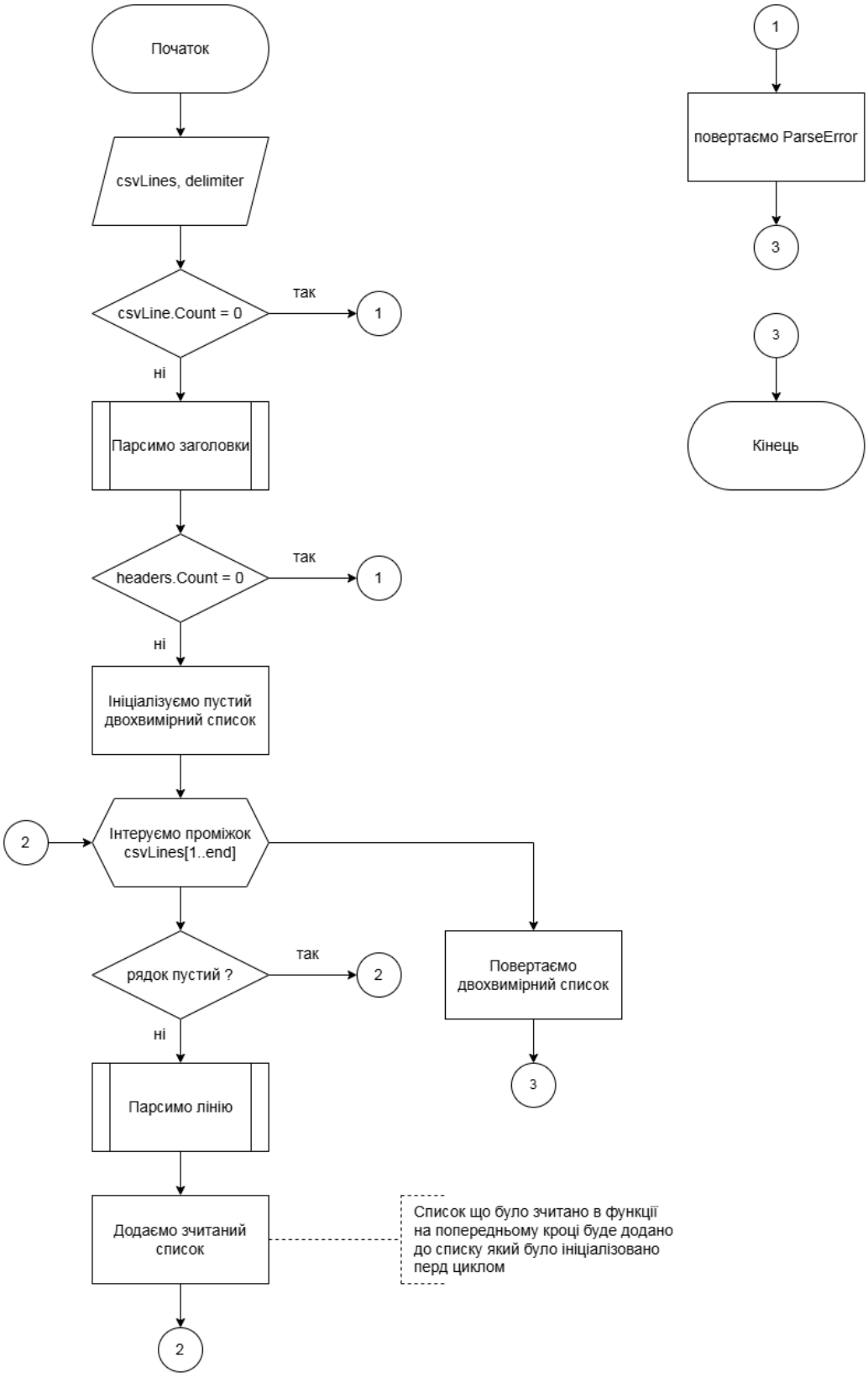

Рис. 24. Блок схема алгоритму для зчитування слів з CSV файлу

Практичну програмну реалізацію наведено на рис. 25. Фінальним результатом роботи цього компонента є формування спеціалізованого об'єкта CsvParseResult, який

містить загальний статус виконання операції, перелік розпізнаних заголовків та згенеровану двовимірну матрицю текстових даних.

```
...
public CsvParseResult ParseCsv(IList<string> csvLines, char delimiter = DefaultDelimiter)
{
    if (csvLines.Count == 0)
    {
        return new CsvParseResult
        {
            IsSuccess = false,
            ErrorMessage = EmptyContentError
        };
    }

    List<string> headers = ParseLine(csvLines[0], delimiter);
    if (headers.Count == 0)
    {
        return new CsvParseResult
        {
            IsSuccess = false,
            ErrorMessage = NoColumnsError
        };
    }

    List<List<string>> dataRows = new List<List<string>>();
    for (int i = 1; i < csvLines.Count; i++)
    {
        if (!string.IsNullOrWhiteSpace(csvLines[i]))
        {
            dataRows.Add(ParseLine(csvLines[i], delimiter));
        }
    }

    return new CsvParseResult
    {
        IsSuccess = true,
        Headers = headers,
        DataRows = dataRows
    };
}
...

public class CsvParseResult
{
    public bool IsSuccess { get; set; }
    public string? ErrorMessage { get; set; }
    public List<string> Headers { get; set; } = new();
    public List<List<string>> DataRows { get; set; } = new();
}
```

Рис. 25. Фрагмент реалізації зчитування слів з CSV файлу

Для забезпечення можливості експорту лексичних баз у формат популярної системи інтервального повторення Anki було розроблено спеціалізований сервіс, програмну реалізацію якого наведено на рис. 26. Даний компонент виконує завантаження структури словника з бази даних та алгоритмічно трансформує її у

структурований текстовий файл, автоматично генеруючи необхідні мета-заголовки колоди, перетворюючи HTML-символи та розділяючи лексичні одиниці й значення табуляцією.

```
public class AnkiExportService : IAnkiExportService
{
    private readonly IRepositoryBase<Vocabulary> _vocabularyRepository;
    public AnkiExportService(IRepositoryBase<Vocabulary> vocabularyRepository)
    {
        _vocabularyRepository = vocabularyRepository;
    }
    public async Task<AnkiExportResult?> ExportAsync(int vocabularyId, CancellationToken ct = default)
    {
        var spec = new VocabularyWithWordsAndMeaningsSpecification(vocabularyId);
        var vocabulary = await _vocabularyRepository.FirstOrDefaultAsync(spec, ct);
        if (vocabulary is null) return null;
        var sb = new StringBuilder();
        sb.Append("#separator:tab\n");
        sb.Append("#html:true\n");
        sb.Append("#notetype:Basic (and reversed card)\n");
        sb.Append($"#deck:{EscapeHeaderValue(vocabulary.Name)}\n");
        foreach (var word in vocabulary.Words)
        {
            var meaningLines = word.Meanings
                .Where(m => !string.IsNullOrEmpty(m.Definition))
                .Select(RenderMeaning)
                .ToList();

            if (meaningLines.Count == 0) continue;
            string front = EscapeHtml(word.WordContent);
            string back = string.Join("<br>", meaningLines);
            sb.Append(front).Append('\t').Append(back).Append('\n');
        }

        return new AnkiExportResult(
            Encoding.UTF8.GetBytes(sb.ToString()),
            SanitizeFileName(vocabulary.Name, vocabularyId));
    }
    private static string RenderMeaning(Meaning m)
    {
        string def = EscapeHtml(m.Definition!);
        return string.IsNullOrEmpty(m.LexemeType)
            ? def
            : $"{def} [{EscapeHtml(m.LexemeType)}]";
    }
    private static string EscapeHtml(string text) =>
        text.Replace("&", "&amp;").Replace("<", "&lt;").Replace(">", "&gt;");
    private static string EscapeHeaderValue(string text) =>
        text.Replace("\n", " ").Replace("\r", " ").Replace("\t", " ");
    private static string SanitizeFileName(string name, int fallbackId)
    {
        var invalid = Path.GetInvalidFileNameChars();
        var sanitized = new string(name.Where(c => !invalid.Contains(c)).ToArray()).Trim();
        return string.IsNullOrEmpty(sanitized)
            ? $"vocabulary-{fallbackId}.txt"
            : $"{sanitized}.txt";
    }
}
```

Рис. 26. Реалізація сервісу для експорту слів в Anki

Висновки до 2-го розділу

У другому розділі було здійснено проєктування та практичну реалізацію адміністративного модуля системи управління цифровими словниками. На основі аналізу вимог було обрано комплексний архітектурний підхід, що поєднує принципи чистої та розподіленої клієнт-серверної архітектури. Таке рішення забезпечило гнучкість, масштабованість та надійну ізоляцію прикладної логіки від зовнішніх інфраструктурних компонентів.

Для програмної реалізації було обрано об'єктно-орієнтовану мову програмування зі строгою статичною типізацією у поєднанні з сучасними інструментами для побудови серверної та клієнтської частин системи. Організацію збереження даних виконано на базі реляційної системи керування базами даних, що гарантує цілісність складної ієрархічної структури лексичних одиниць. Проєктування схеми бази даних здійснено за методологією первинності коду, що дозволило автоматизувати процеси відображення об'єктних моделей у реляційні таблиці.

З метою забезпечення безпеки інформації та розмежування прав доступу до програмного комплексу було обрано незалежну підсистему управління ідентифікацією для інтеграцію з системою.

Окрім архітектурних рішень, у розділі реалізовано ключові функціональні можливості модуля. Для прискорення процесів наповнення баз даних було розроблено алгоритми автоматизованого вилучення інформації шляхом синтаксичного аналізу зовнішніх веб сторінок, а також механізми імпортування даних зі структурованих текстових файлів. Додатково було реалізовано підсистему експортування, яка здійснює алгоритмічне перетворення сформованих словників у стандартизований формат для забезпечення їх сумісності зі сторонніми навчальними платформами. У результаті, розроблені інженерні та програмні рішення дозволили створити надійний інструмент, який повною мірою задовольняє потреби адміністрування навчальної лексики.

РОЗДІЛ 3

СИСТЕМНА ІНТЕГРАЦІЯ ТА ПЕРЕВІРКА ФУНКЦІОНАЛУ

3.1 Контейнеризація та розгортання системи

Сучасна парадигма розробки розподілених інформаційних систем об'єктивно потребує забезпечення ідентичності та стабільності середовищ виконання незалежно від характеристик цільової інфраструктури. Ефективним архітектурним рішенням для автоматизації процесів розгортання та ізоляції компонентів є застосування технології контейнеризації, яка дозволяє розмістити виконуваний код разом з усіма необхідними залежностями. У межах розробленого проєкту платформу Docker та інструментарій Docker Compose використано для управління повноцінного локального програмного стеку, що органічно доповнює обраний підхід розподіленої архітектури.

Розгорнута інфраструктура об'єднує в єдиній віртуальній мережі реляційну базу даних PostgreSQL, сервер управління ідентифікацією Keycloak, програмний інтерфейс ASP.NET Core Web API та клієнтський застосунок Blazor WebAssembly, статичні файли якого обслуговуються веб сервером Nginx. Завдяки застосуванню оптимізованих багатоетапних збірок для генерації власних образів серверної та клієнтської частин, система досягає мінімізації розміру кінцевих контейнерів, ефективно усуває конфлікти конфігурацій та гарантує повністю автоматизований і передбачуваний запуск усього комплексу взаємопов'язаних сервісів.

Для оптимізації процесу розгортання серверного програмного інтерфейсу застосовано підхід багатоетапного збирання, програмну конфігурацію якого наведено на рис. 27. На початковому етапі в базовому середовищі виконується відновлення залежностей, компіляція та формалізована публікація вихідного коду API. Фінальний етап формує мінімально необхідне середовище виконання куди переносяться

виключно готові бінарні файли. Така структуризація дозволяє відсіяти надлишкові інструменти розробки та суттєво зменшити розмір кінцевого Docker-образу API, підвищуючи рівень безпеки та загальну продуктивність компонента.

Для розгортання клієнтського застосунку на базі Blazor WebAssembly використано споріднену стратегію багатоетапного збирання, конфігурацію якої проілюстровано на рис. 28. Зважаючи на архітектурні особливості обраного фреймворку, результатом компіляції виступає набір статичних файлів, тому оптимізоване середовище виконання базується на легкому та ефективному веб сервері Nginx, а не на повноцінному середовищі .NET. Після збирання проєкту на етапі SDK, до фінального контейнера переноситься виключно згенерований статичний контент разом із відповідним конфігураційним файлом сервера для коректної маршрутизації. Таке рішення гарантує швидку, ізольовану та ефективну роздачу користувацького інтерфейсу без необхідності залучення надлишкових обчислювальних платформ.

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

COPY src/VocabularyManager.Api/VocabularyManager.Api.csproj src/VocabularyManager.Api/
COPY src/VocabularyManager.Core/VocabularyManager.Core.csproj src/VocabularyManager.Core/
COPY src/VocabularyManager.Infrastructure/VocabularyManager.Infrastructure.csproj src/VocabularyManager.Infrastructure/
COPY src/VocabularyManager.UseCases/VocabularyManager.UseCases.csproj src/VocabularyManager.UseCases/

RUN dotnet restore src/VocabularyManager.Api/VocabularyManager.Api.csproj

COPY src/ src/

RUN dotnet publish src/VocabularyManager.Api/VocabularyManager.Api.csproj \
    -c Release \
    -o /app/publish \
    --no-restore

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
RUN apt-get update && apt-get install -y --no-install-recommends curl && rm -rf /var/lib/apt/lists/*
WORKDIR /app
COPY --from=build /app/publish .
EXPOSE 8080
ENV ASPNETCORE_URLS=http://+:8080
ENTRYPOINT ["dotnet", "VocabularyManager.Api.dll"]
```

Рис. 27. Реалізація Docker-образу для API

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

COPY src/VocabularyManager.BlazorApp/VocabularyManager.BlazorApp.csproj src/VocabularyManager.BlazorApp/
COPY src/VocabularyManager.Core/VocabularyManager.Core.csproj src/VocabularyManager.Core/
COPY src/VocabularyManager.UseCases/VocabularyManager.UseCases.csproj src/VocabularyManager.UseCases/

RUN dotnet restore src/VocabularyManager.BlazorApp/VocabularyManager.BlazorApp.csproj

COPY src/ src/

RUN dotnet publish src/VocabularyManager.BlazorApp/VocabularyManager.BlazorApp.csproj \
    -c Release \
    -o /app/publish \
    --no-restore

FROM nginx:alpine AS runtime
COPY --from=build /app/publish/wwwroot /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
```

Рис. 28. Реалізація Docker-образу клієнтської частини

Зважаючи на розподілену архітектуру розробленої системи, яка об'єднує низку незалежних контейнерів, виникає об'єктивна необхідність у централізованому управлінні. Для автоматизації процесу комплексного розгортання, налаштування взаємодії у локальній мережі, безпечної передачі змінних середовища та контролю послідовності запуску залежних сервісів було застосовано інструментарій Docker Compose.

В результаті дана реалізація сформувала цілісну, стійку та гнучку інфраструктуру програмного комплексу. Такий архітектурний підхід забезпечує повну автоматизацію процесів конфігурації та комплексного розгортання всіх взаємозалежних модулів системи, включаючи реляційну базу даних, сервер ідентифікації Keycloak, програмний інтерфейс та клієнтський веб застосунок. При цьому розгортання суцільного комплексу інфраструктури виконується одною командою «docker compose up -build -d».

3.2 Автоматизоване тестування

Забезпечення високої надійності та безперебійного функціонування розробленої розподіленої системи управління словниками неможливе без

впровадження комплексного підходу до перевірки її компонентів. Зі зростанням функціональної складності програмного забезпечення, розширенням бізнес-логіки та наявністю багаторівневих інтеграцій, традиційна ручна перевірка стає вкрай неефективною та схильною до людських помилок. Саме тому виникає об'єктивна необхідність у тестуванні програмного забезпечення, з особливим акцентом на автоматизації цього процесу. Створення автоматизованих тестів дозволяє регулярно перевіряти працездатність коду та швидко виявляти дефекти при внесені змін. Варто зазначити, що закладений раніше підхід чистої архітектури безпосередньо сприяє побудові системи, що легко тестується, та суттєво спрощує процес автоматизованої перевірки її ключових компонентів [42].

Модульне тестування (англ. *Unit Testing*) — це рівень автоматизованої перевірки програмного забезпечення, під час якого перевіряється коректність функціонування найменших неподільних та програмно ізольованих фрагментів вихідного коду (таких як окремі класи чи методи). Головна мета цього процесу полягає у підтвердженні того, що кожна окрема програмна одиниця працює точно відповідно до закладеної специфікації та ізольовано від залежностей [42].

Для забезпечення стандартизації, високої читабельності та зручності підтримки автоматизованих тестів у розробленому модулі було застосовано загальноприйнятий архітектурний шаблон AAA (Arrange-Act-Assert). Цей методологічний підхід передбачає логічну декомпозицію кожного тестового сценарію на три послідовні фази. На етапі підготовки (Arrange) здійснюється ініціалізація необхідних об'єктів, налаштування заглушок для ізоляції зовнішніх залежностей та визначення вхідних параметрів. Етап виконання (Act) передбачає безпосередній виклик цільового методу або функції, яка підлягає перевірці. Завершальна фаза перевірки (Assert) полягає у формальному порівнянні отриманого результату виконання або зміненого стану системи з очікуваними результатами, що дозволяє зробити об'єктивний висновок щодо коректності роботи досліджуваної програмної одиниці [42].

Для забезпечення коректності роботи механізмів оновлення лексичних даних було розроблено відповідні автоматизовані сценарії перевірки. Зокрема, програмну реалізацію модульного тесту, який перевіряє логіку додавання значень до існуючого

слова (рис. 28). Головна мета цього сценарію полягає у підтвердженні того, що під час спроби збереження списку значень, який містить як нові, так і вже існуючі в базі даних елементи, система правильно ідентифікує та відфільтровує дублікати, інтегруючи до колекції виключно нові записи. Такий алгоритм роботи гарантує цілісність інформації та запобігає виникненню надлишковості даних у межах словника.

```
[Fact]
public async Task AddMeanings_WhenMixOfNewAndDuplicate_ShouldAddOnlyNew()
{
    // Arrange
    int wordId = _fixture.Create<int>();
    Meaning existingMeaning = new Meaning("noun", "existing definition", "A1") { WordId = wordId };
    Word existingWord = new Word("test")
    {
        Id = wordId,
        Meanings = new List<Meaning> { existingMeaning }
    };

    _wordRepository
        .FirstOrDefaultAsync(Arg.Any<ISpecification<Word>>())
        .Returns(Task.FromResult<Word?>(existingWord));

    List<Meaning> meanings = new List<Meaning>
    {
        new Meaning("noun", "existing definition", "A1"),
        new Meaning("verb", "new definition", "B2")
    };

    // Act
    IEnumerable<int> result = await _meaningStorageManager.AddMeanings(meanings, wordId);

    // Assert
    result.Should().HaveCount(1);
    await _meaningRepository.Received(1).AddAsync(Arg.Any<Meaning>());
}
```

Рис. 28. Приклад сценарію тестування для операції додавання значень слів

На етапі підготовки (Arrange) здійснюється генерація ідентифікатора, ініціалізація сутності слова з наявним тестовим значенням, налаштування заглушки репозиторію для імітації стану бази даних, а також формування вхідної колекції, що містить нове та дубльоване значення. На етапі виконання (Act) виконується безпосередній виклик досліджуваного методу збереження AddMeanings. Завершальна фаза перевірки (Assert) за допомогою спеціалізованих тверджень підтверджує, що колекція результатів, яка повертається методом (відображаючи кількість успішно доданих записів), містить лише один елемент, а інструментарій заглушок перевіряє, що метод додавання до репозиторію був викликаний рівно один раз. Це доводить, що компонент обробив та зберіг лише унікальне значення, успішно проігнорувавши дублікат.

Шаблон проєктування «Прототип», що полягає у створенні нових об'єктів шляхом клонування існуючих, надзвичайно часто застосовується для швидкої та оптимізованої ініціалізації складних структур даних. Оскільки такий механізм зазвичай стає фундаментальним компонентом системи, від якого залежить подальша бізнес-логіка, його ретельне покриття тестами є критично необхідним. Найменша неточність у реалізації (наприклад, копіювання за посиланням замість створення незалежного об'єкта) здатна спровокувати каскадні збої, тому автоматизована перевірка цього кореневого функціоналу є базовою умовою стабільності всього застосунку.

Для верифікації коректності роботи механізму клонування слів було розроблено набір модульних тестів. Вони використовують підхід AAA та інструментарій генерації випадкових даних, покриваючи наступні сценарії, які наведені на рис. 29.

```
public class WordClonerTests
{
    Fixture _fixture;
    public WordClonerTests()
    {
        _fixture = new Fixture();
    }

    [Fact]
    public void Word_Clone_Test1()
    {
        //Arrange
        var meaning = new MeaningView(
            _fixture.Create<string?>(),
            _fixture.Create<string?>(),
            _fixture.Create<string?>()
        );
        var word = new WordView(_fixture.Create<string>(), new List<MeaningView> { meaning });

        //Act
        var clonedWord = WordCloner.CloneWord(word);

        //Assert
        clonedWord
            .Should().NotBeSameAs(word);
        clonedWord
            .Should().BeEquivalentTo(word);
    }

    [Fact]
    public void Word_Clone_Without_Meanings_Test()
    {
        //Arrange
        var word = new WordView(_fixture.Create<string>());

        //Act
        var clonedWord = WordCloner.CloneWord(word);

        //Assert
        clonedWord
            .Should().NotBeSameAs(word);
        clonedWord
            .Should().BeEquivalentTo(word);
    }
}
```

Рис. 29. Набір тестів для реалізації дизайн шаблону «Прототип»

Перший тестовий сценарій спрямований на перевірку базової операції клонування об'єкта з повною структурою вкладених даних. Його головне завдання полягає у верифікації здатності компонента правильно копіювати комплексну сутність слова, яка містить заповнену пов'язану колекцію значень. Цей тест доводить, що алгоритм здатний правильно відтворювати всі існуючі рівні ієрархії складного об'єкта, зберігаючи його цілісність.

Другий тестовий сценарій аналізує процес клонування базового об'єкта в умовах повної відсутності вкладених сутностей. Його основна мета – переконатися, що логіка системи є стійкою до неповних даних, зокрема коли сутність слова заповнена, але ще не має жодного доданого значення. Така перевірка гарантує безпечну поведінку алгоритму під час обробки специфічних станів доменної моделі.

З точки зору програмної реалізації та потоку виконання, перший сценарій конструює комплексний об'єкт на етапі підготовки та викликає метод клонування, після чого переходить до строгої фази перевірки (Assert). На цьому етапі використовуються два ключові твердження: метод NotBeSameAs гарантує, що створений клон є абсолютно новим та незалежним об'єктом у пам'яті (відсутність спільного посилання з оригіналом). В своєю чергу, метод BeEquivalentTo підтверджує повну ідентичність їхнього внутрішнього стану, фіксуючи точний збіг значень усіх властивостей оригінального та скопійованого об'єктів.

Процес виконання другого сценарію розпочинається зі створення спрощеного об'єкта-оригіналу без виділення пам'яті під колекцію значень. Потік виконання передає цей об'єкт до механізму клонування, перевіряючи, чи компонент правильно обробляє порожні колекції або null-посилання без виникнення помилок часу виконання (на кшталт NullReferenceException). Фаза перевірки завершується застосуванням аналогічних тверджень, які підтверджують, що система успішно згенерувала структурно ідентичну, але цілком незалежну копію базового об'єкта навіть за умов відсутності вкладених даних.

У ході роботи було застосовано підхід автоматизованого тестування, який є невід'ємною складовою розробки надійного програмного забезпечення. Цей метод

дозволив виконувати перевірку значної кількості різноманітних тестових сценаріїв у межах часток секунди.

На рис. 30 наведено результат запуску усього комплексу з 38 модульних тестів, який виконується натисканням однієї кнопки в інтегрованому середовищі розробки. Можна побачити, що всі сценарії тестування виконуються успішно, отже функціонал покритий тестами можна вважати реалізованим правильно. Це значно спростило та пришвидшило рутинний процес верифікації. Завдяки цьому було забезпечено миттєвий зворотний зв'язок щодо стану системи, мінімізовано витрати часу на ручне тестування та гарантовано стабільність коду під час його подальшого масштабування чи модифікації.

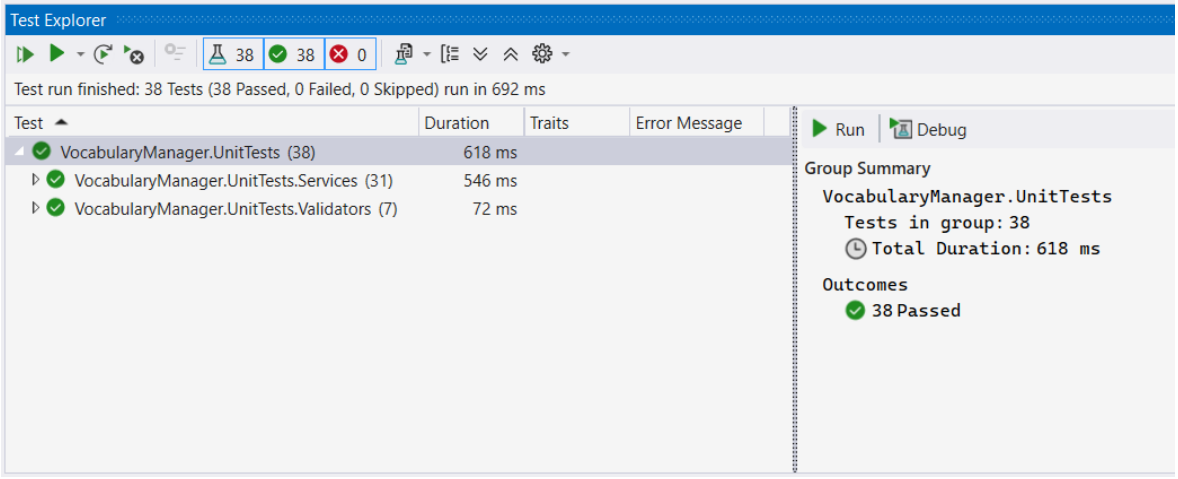

Рис. 30. Результат виконання автоматизованих тестів

3.3 Демонстрація функціоналу

Для наочної демонстрації практичних результатів роботи розробленого адміністративного модуля та його графічного інтерфейсу доцільно розглянути базові сценарії взаємодії користувача з системою. Оскільки безпека та суворе розмежування прав доступу є фундаментальними вимогами до програмного комплексу, користувацький досвід зазвичай розпочинається з етапу електронної ідентифікації.

На рис. 31 зображено сторінку реєстрації та автентифікації, процеси яких повністю делеговані інтегрованому серверу авторизації Keycloak.

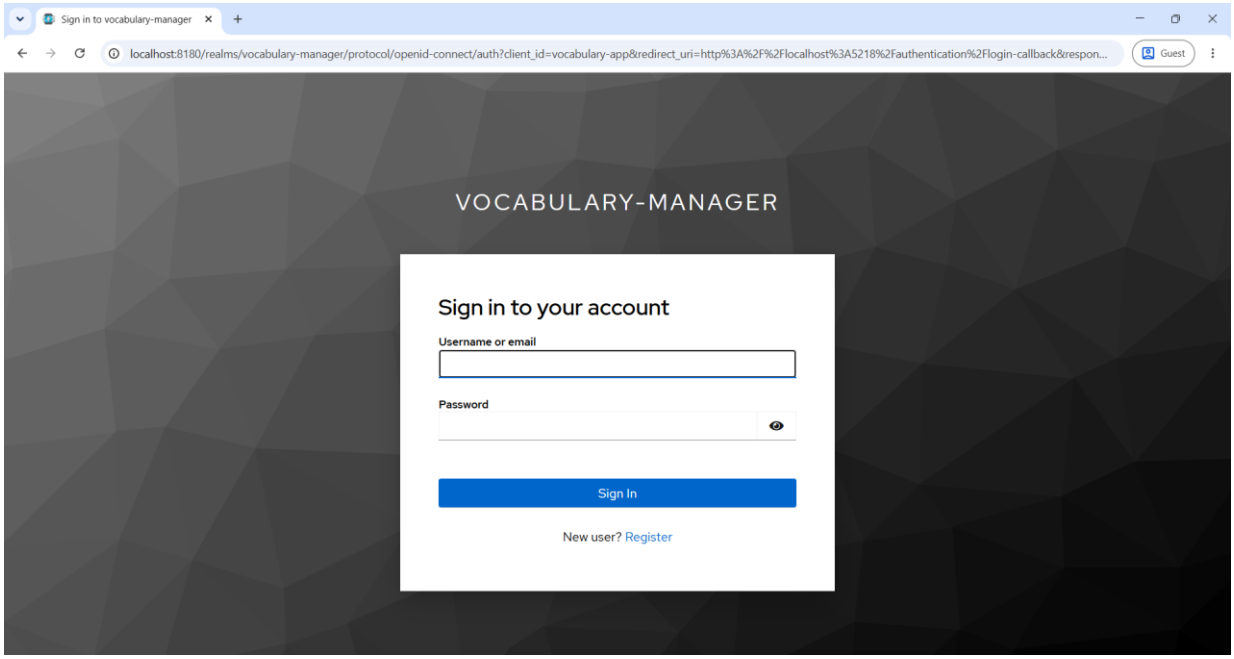

Рис. 31. Вигляд сторінки для входу в систему

Після успішної операції автентифікації користувач буде направлений на головну сторінку клієнтської частини системи, яка зображена на рис. 32.

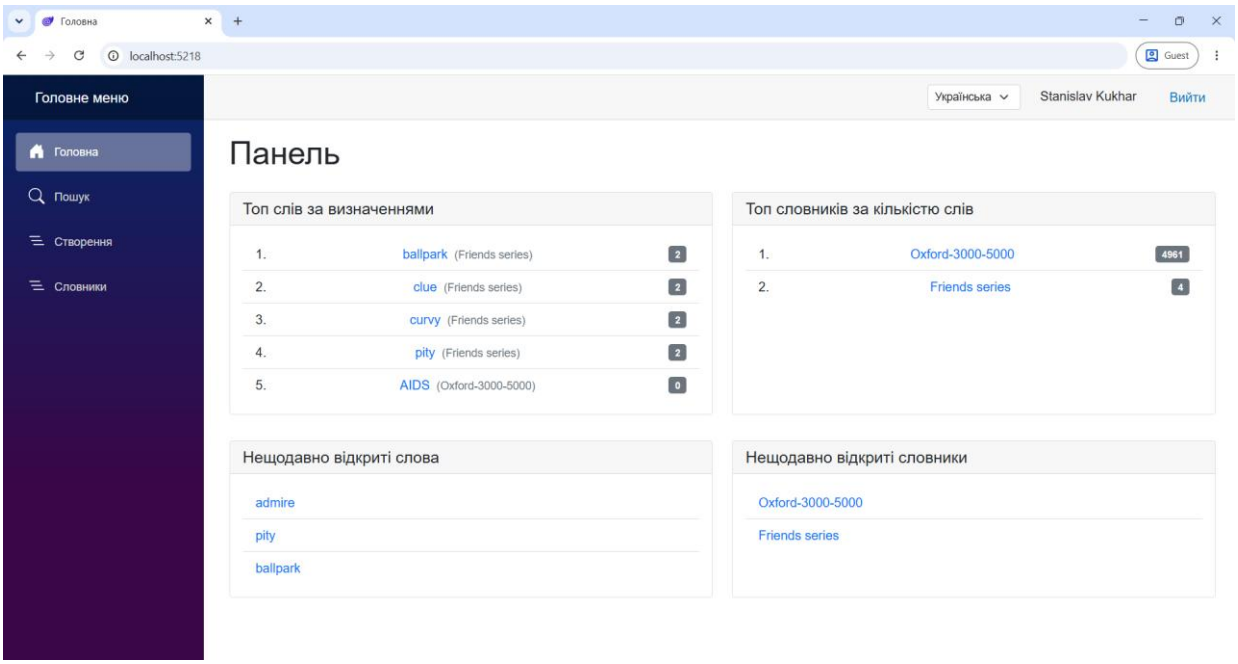

Рис. 32. Вигляд головної сторінки клієнтської частини

Архітектура користувацького інтерфейсу має чітку структуру, яку можна розділити на три основні функціональні компоненти:

- Бічну навігаційну панель яка розташована ліворуч. Панель забезпечує глобальне орієнтування та зручне переміщення між модулями системи.
- Верхню панель, яка містить елементи глобального управління: інструмент перемикання мови, кнопку завершення сесії та блок відображення імені поточного користувача.
- Центральну робочу область, контент якої оновлюється залежно від поточного стану застосунку або обраного функціонального розділу.

На головній сторінці застосунку розташовано чотири функціональні блоки, призначені для відображення статистичних даних та оптимізації навігації користувача:

- «Топ слів за визначеннями»: блок відображає рейтинг лексичних одиниць із найбільшою кількістю закріплених за ними значень. Для кожної позиції вказано назву слова, відповідний словник та числовий показник кількості значень.
- «Топ словників за кількістю слів»: аналітична панель, що представляє статистичну вибірку наявних словникових баз, впорядкованих за загальним обсягом лексичних одиниць.
- «Нещодавно відкриті слова»: елемент швидкого доступу, що фіксує історію взаємодії та у вигляді списку відображає останні переглянуті користувачем слова.
- «Нещодавно відкриті словники»: панель навігації, яка містить перелік останніх активних словників для забезпечення швидкого повернення до попередніх робочих сесій.

Сторінка пошуку відповідає за зручну та швидку орієнтацію у словникових базах. Сторінка має три основні компоненти взаємодії, що зображені на рис. 33:

- Поле фільтра слова та список результатів, які дозволяють знайти бажане слово, вказуючи набір літер чи частину слова.

- Поле фільтра словників, яке дозволяє звузити чи розширити область пошуку слів.
- Перемикач «глобального режиму», який дозволяє відображати значення для конкретного слова незалежно від кількості значень, розподілених по різних словниках. За умови ввімкненого перемикача результат пошуку відобразить значення для конкретного слова, об’єднуючи результати пошуку по всій базі.

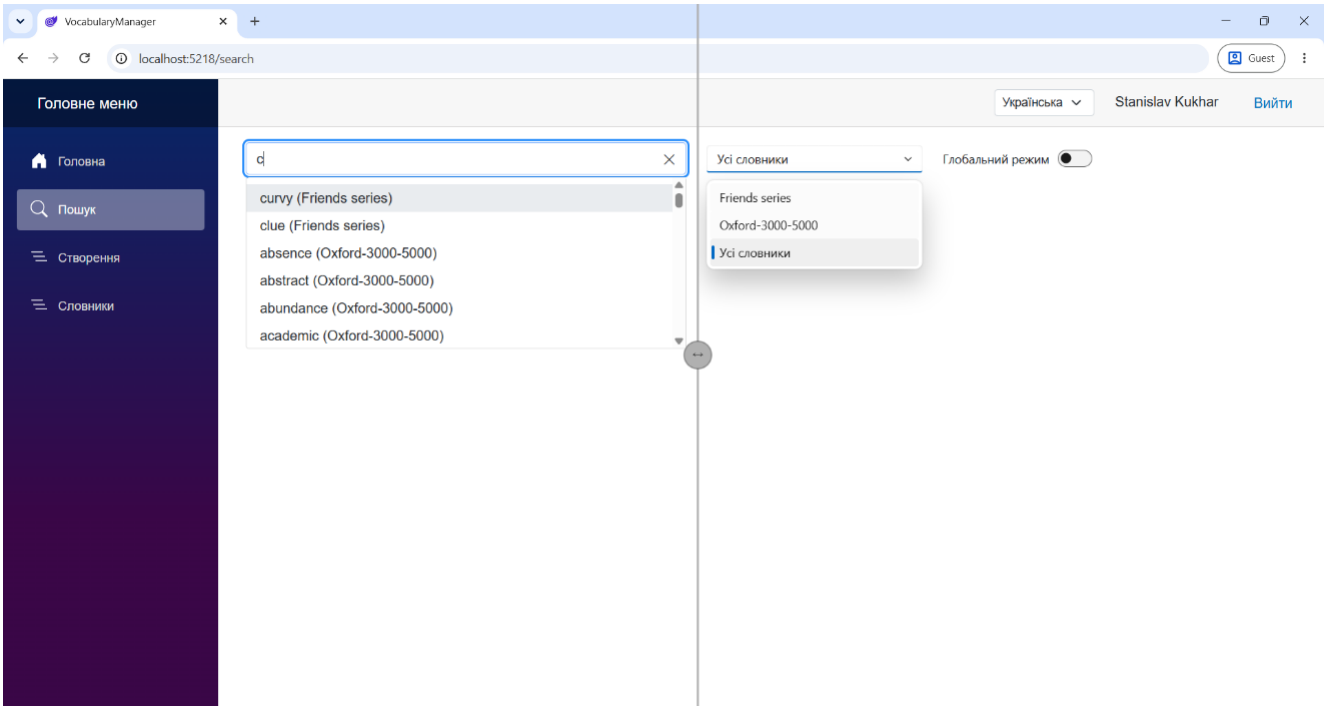

Рис 33. Сторінка пошуку слів

На рис. 34 наведено приклад використання функціоналу для імпортування слів із CSV-файлу. Процес імпортування розпочинається з вибору фізичного файлу зі сховища кінцевого пристрою. Після цього користувач може встановити відповідність між полями у файлі та полями сутності (атрибутами таблиці). Кінцевим етапом є застосування операції до конкретного словника, яким може бути вже існуючий або новий, що буде створений під час імпортування слів. Функціонал для збереження відображається під час взаємодії з кнопкою «Застосувати зіставлення». Користувацький інтерфейс якого наведено на рис. 35.

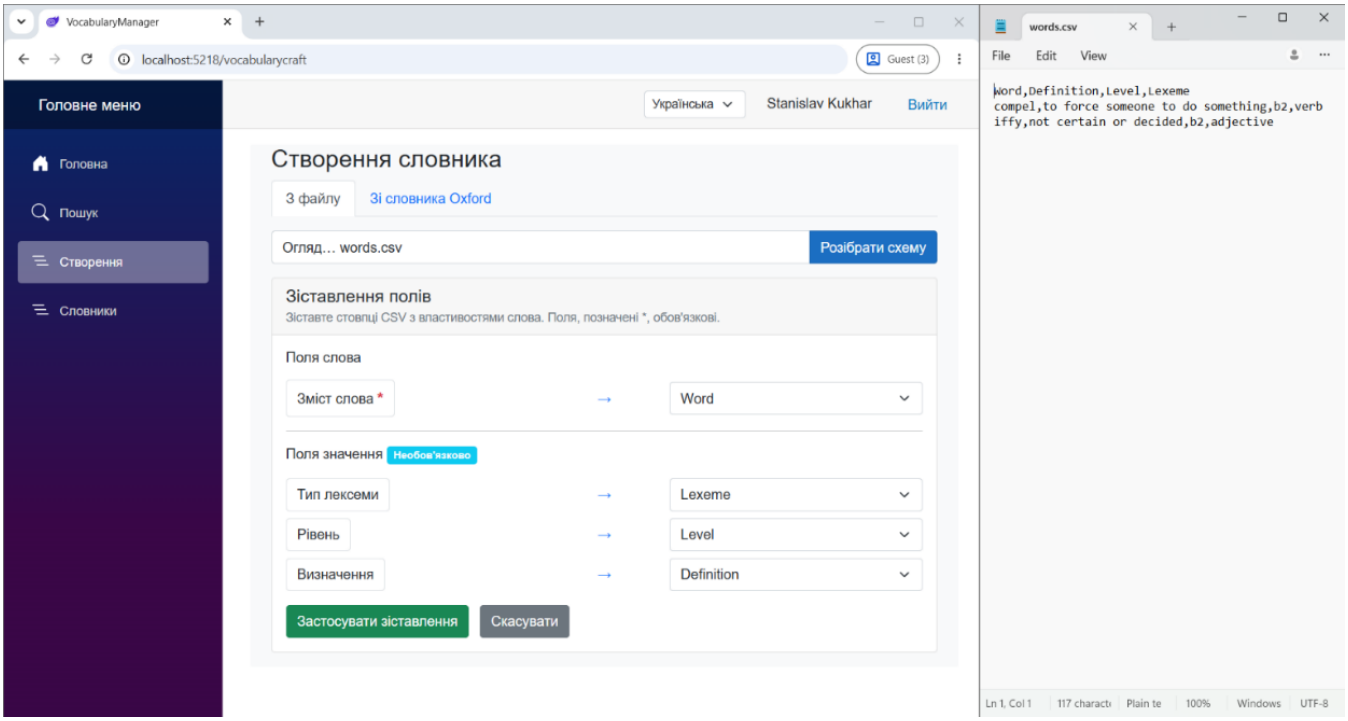

Рис. 34. Функціонал імпортування слів з CSV файлів

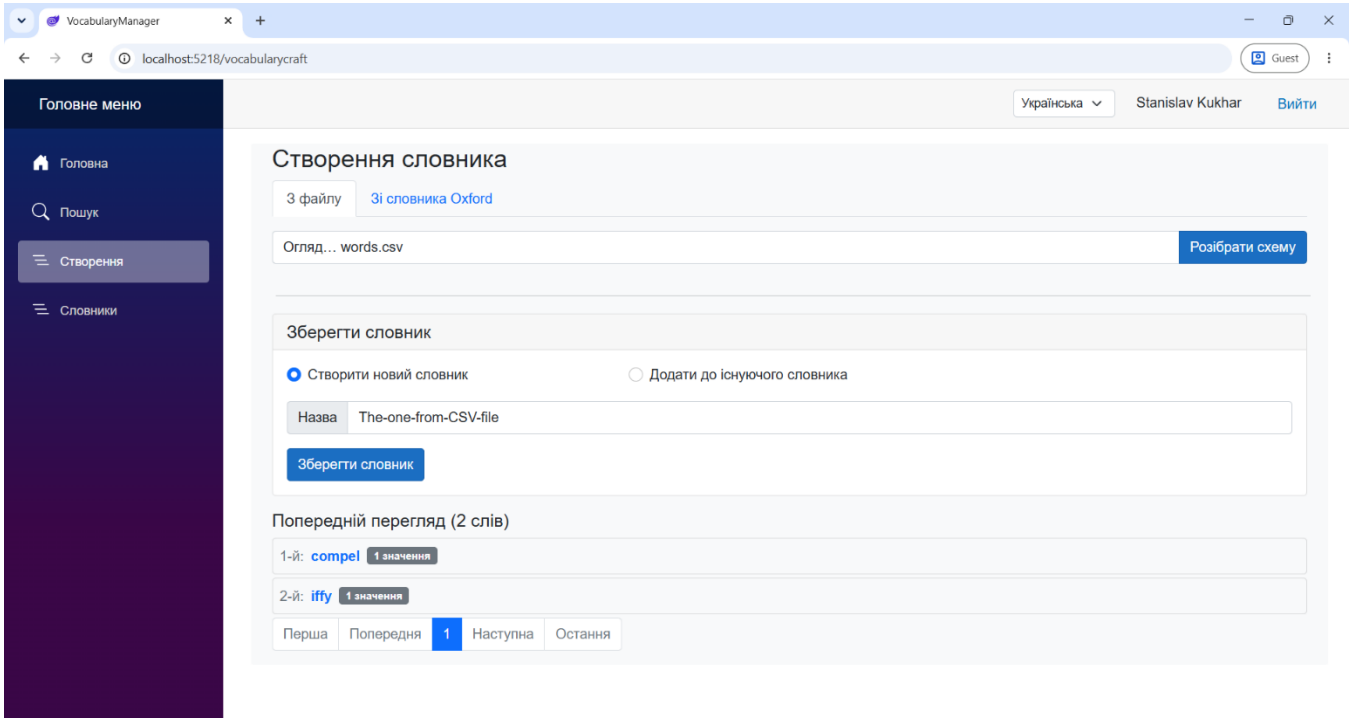

Рис. 35. Функціонал для збереження слів імпортованих з CSV файлу

Попри імпортування слів з CSV файлів система дозволяє імпортувати слова з oxford словника за посиланням на веб сторінку. Даний функціонал наведено на рис. 36.

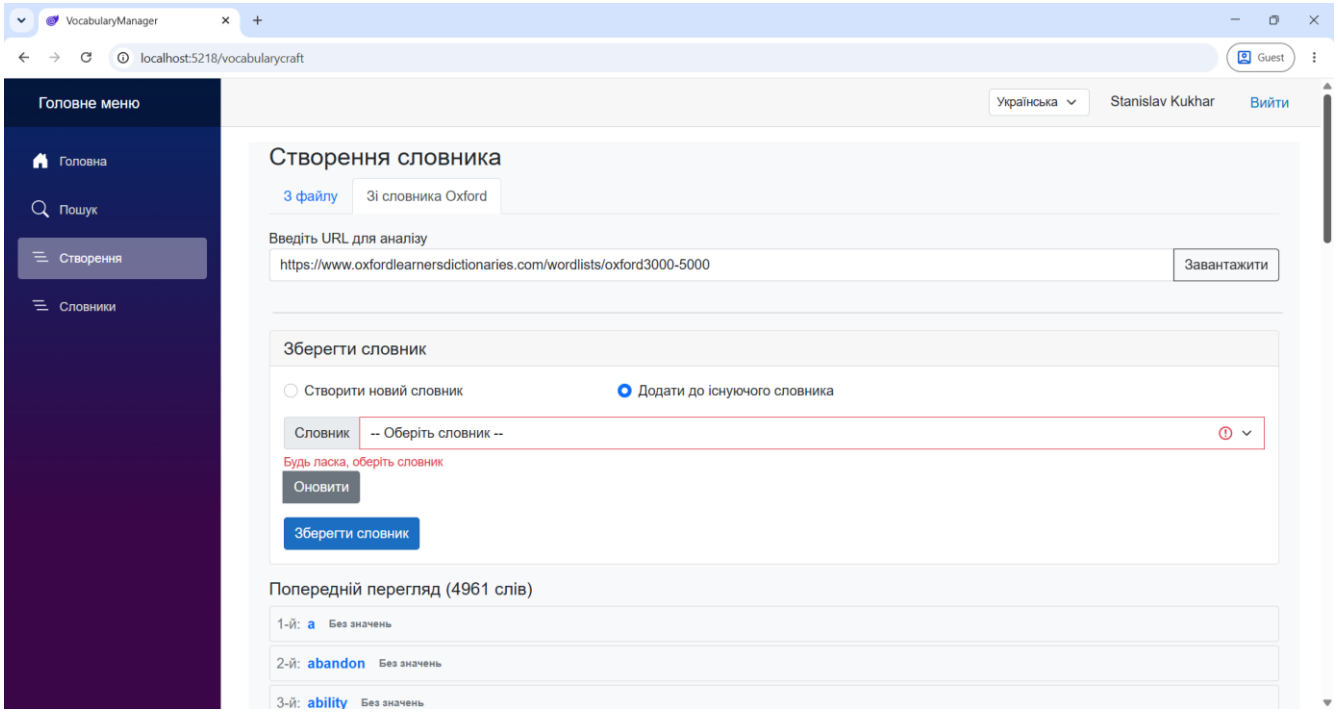

Рис. 36. Функціонал імпортування слів з oxford словника

Сторінка словників містить поле для пошуку словників за іменем та список словників з елементами фільтрації та навігації по сторінках. Даний функціонал наведено на рис. 37.

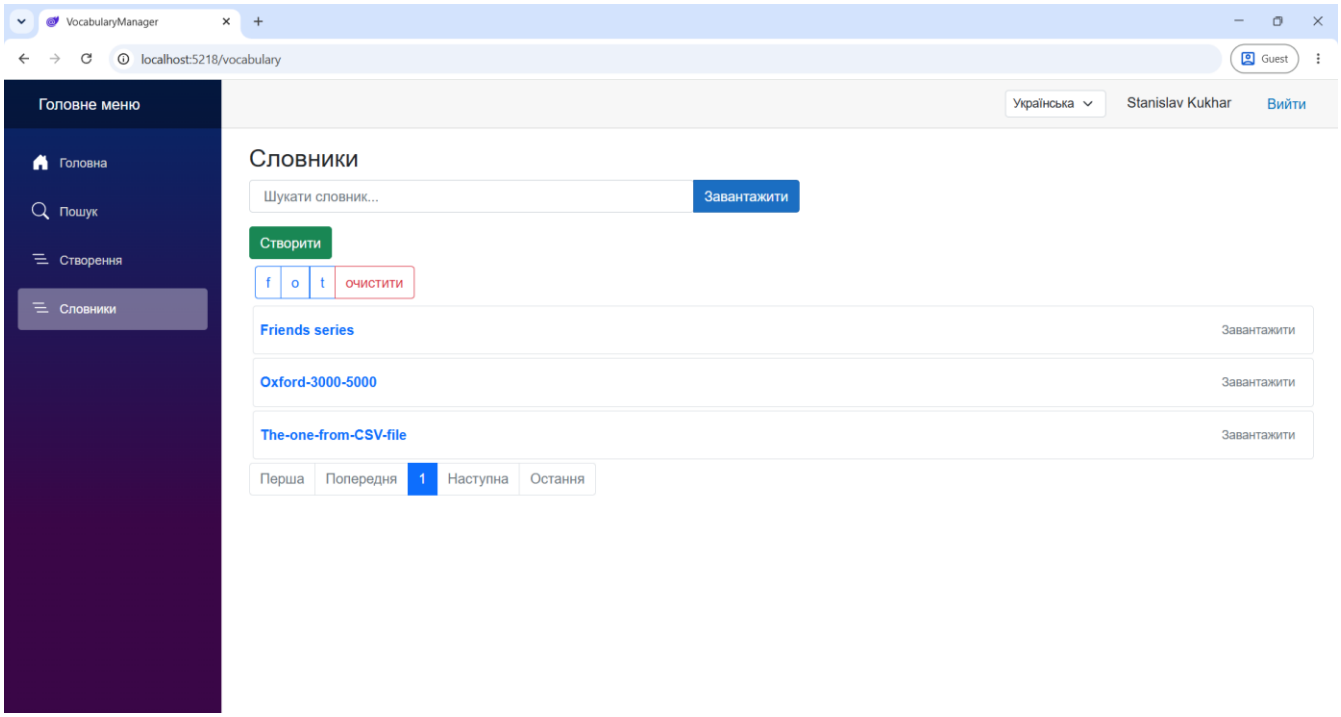

Рис. 37. Вигляд сторінки словників

На рис. 38 зображено вигляд сторінки після вибору словника. Процес вибору словника відбувається шляхом натискання на елемент у списку словників. Інтерфейс має декілька основних елементів керування. Кнопка «Створити» використовується для створення власного словника. Кнопка вивантаження закриває словник та повертає інтерфейс користувача до стану переліку словників, зображеного на рис. 37. Кнопка «Експортувати в Anki» дозволяє сформувати та завантажити текстовий файл словника, який може бути імпортований у систему Anki. Остання кнопка червоного кольору призначена для видалення словника з бази.

Крім елементів керування у вигляді кнопок, сторінка також має дві вкладки. Активна вкладка «Про словник» містить загальну інформацію про словник: назву, кількість слів та джерело словника (за умови, якщо воно було зазначено під час створення). Вкладка «Слова» дозволяє працювати зі словами словника безпосередньо. Вигляд вкладки наведено на рис. 39. Додатково цей рисунок підтверджує роботу функціоналу імпортування слів із CSV-файлу, що було продемонстровано на рис. 35.

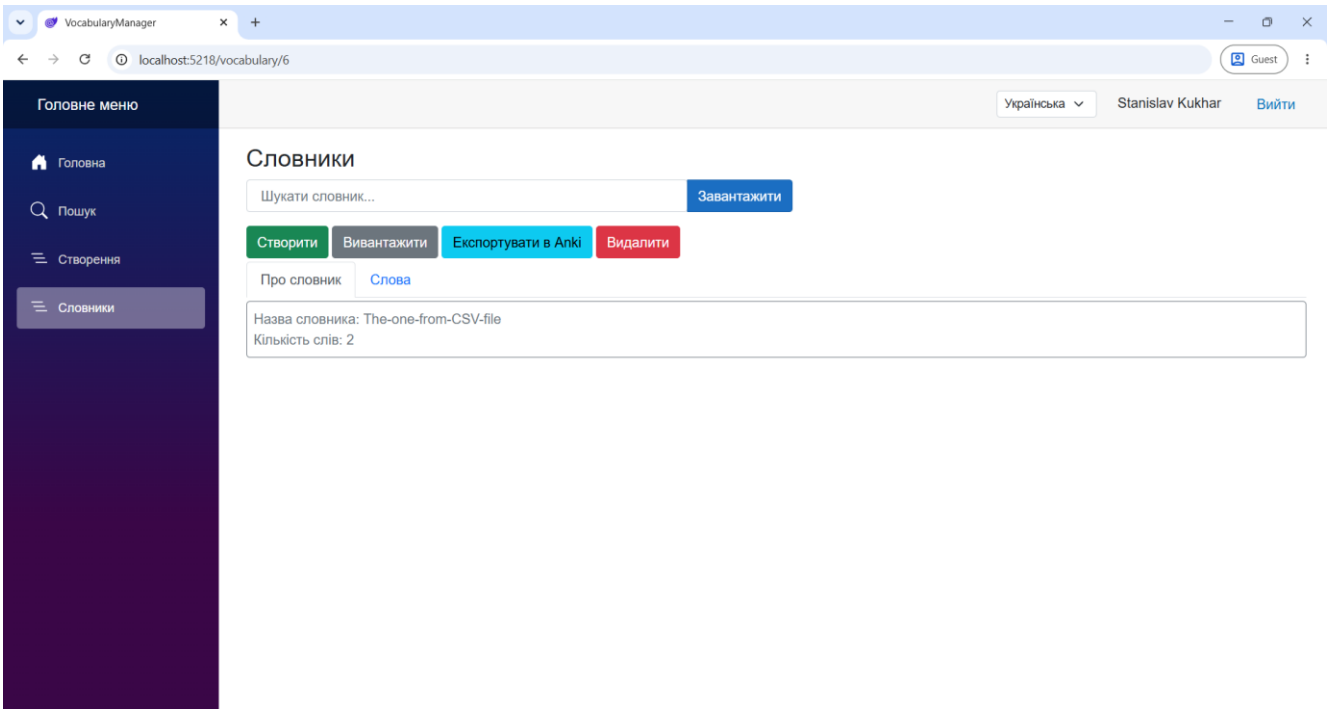

Рис. 38. Вкладка загальної інформації словника

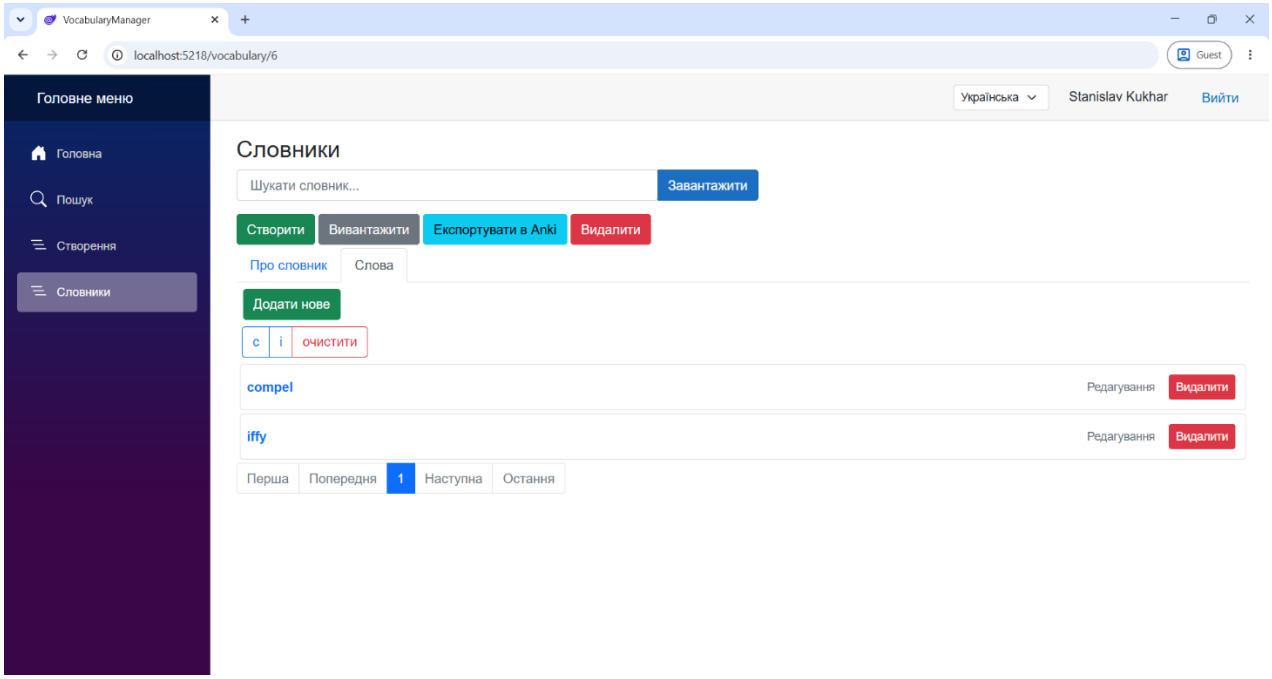

Рис. 39. Вигляд сторінки слів конкретного словника

На рис. 40 зображено результат експортування словника в форматі, що підтримується системою Anki. Для імпортування потрібно створити мати існуючий словник в системі Anki. Після потрібно використати функціонал системи зображений на рис. 41, посилаючись на файл який було звантажено на етапі експортування словника. Результат процесу імпорту наведено на рис. 42.

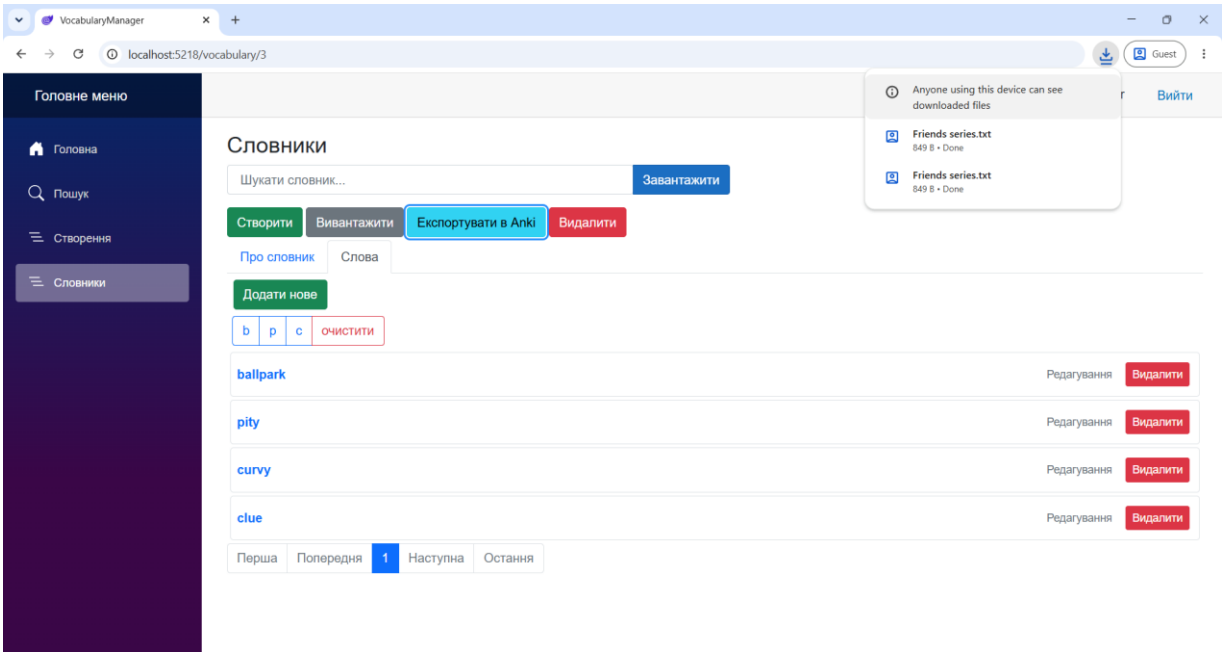

Рис. 40. Результат експорту словника

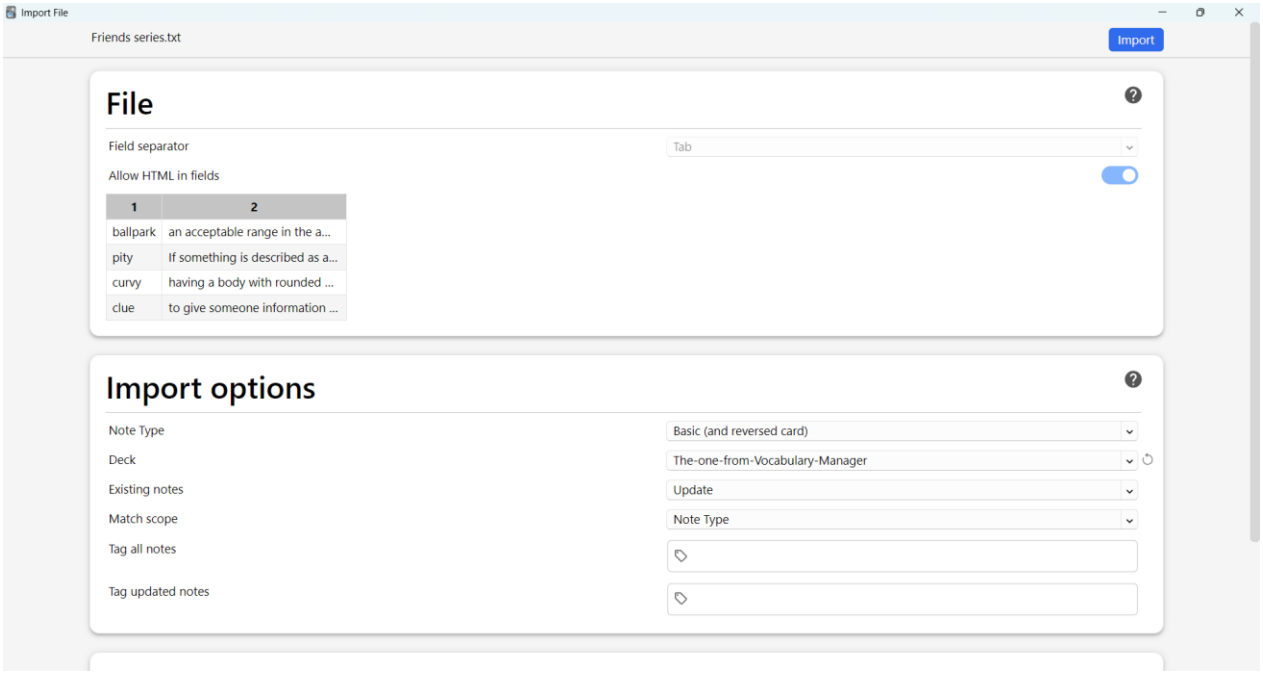

Рис 41. Процес імпорту експортованого словника в систему Anki

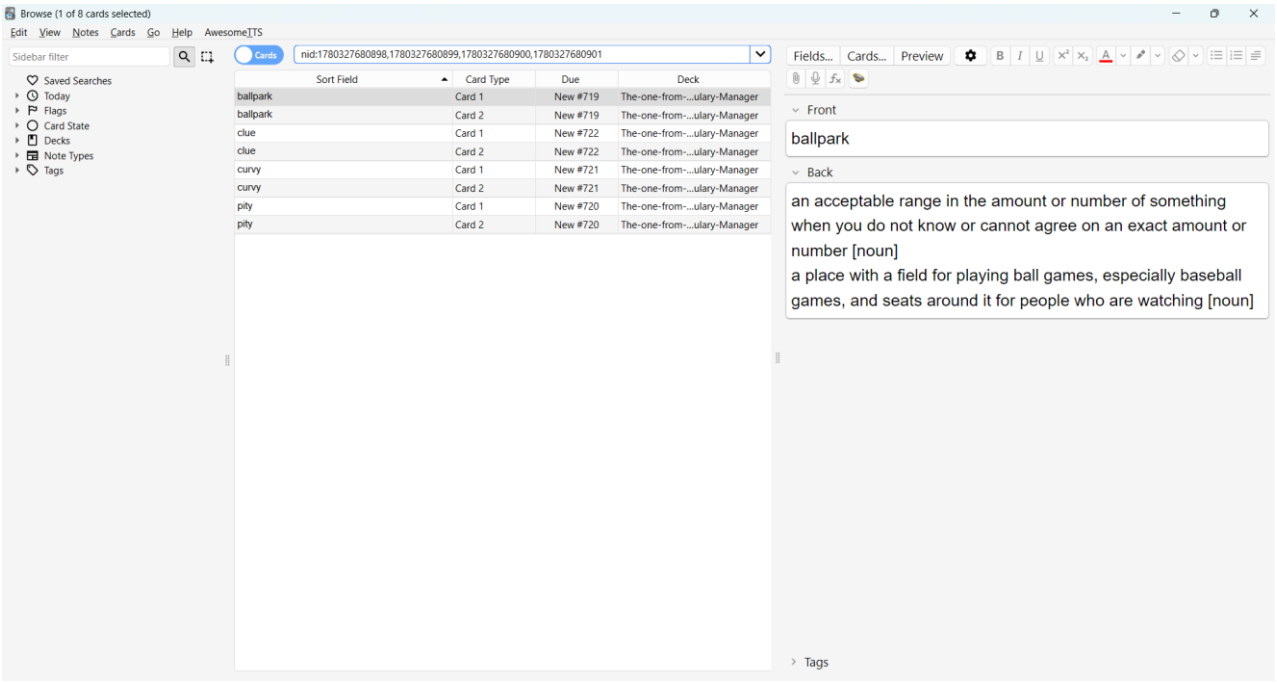

Рис. 42. Результат операції імпорту словника в системі Anki

Висновки до 3-го розділу

В роботі було здійснено системну інтеграцію розробленого програмного комплексу та комплексну перевірку його функціональних можливостей. Для забезпечення надійного та стабільного функціонування застосовано технологію ізоляції процесів у віртуальних середовищах, що дозволило об'єднати всі складові у єдину інфраструктуру. Такий архітектурний крок гарантував повну автоматизацію процесів розгортання, усунення конфліктів конфігурацій та незалежність роботи модуля від апаратних характеристик цільового середовища виконання.

З метою підтвердження високої надійності алгоритмів та безперебійності прикладної логіки було впроваджено комплекс автоматизованих тестів, а саме модульних тестів. Це забезпечило надійний механізм швидкого виявлення потенційних дефектів та заклало міцну основу для подальшого безпечного розширення програмного рішення.

Окрім інфраструктурних налаштувань, у розділі наочно продемонстровано практичні результати роботи розробленого графічного інтерфейсу користувача. Детально описано базові сценарії взаємодії із системою, починаючи від етапу електронної ідентифікації та завершуючи складними процесами управління лексичним контентом. Практично підтверджено коректність роботи аналітичних панелей, інструментів розширеного наскрізного пошуку, механізмів автоматизованого завантаження словникових баз із зовнішніх мережевих ресурсів та структурованих текстових файлів, а також функцій перетворення та вивантаження даних у стандартизовані формати для сторонніх систем навчання.

Після інтеграції системних компонентів та налаштування інфраструктури було підтверджено повну працездатність розробленого адміністративного модуля.

ВИСНОВКИ

При виконанні дипломної роботи було проведено аналіз поставленого завдання, предметної області, існуючих рішень, проблем області та встановлено вимоги до адміністративного модуля.

Під час аналізу було визначено, що процес опанування іноземних мов неминуче пов’язаний із необхідністю засвоєння великих масивів лексики, обсяг якої може сягати десятків тисяч одиниць. Використання науково обґрунтованих методів, таких як системи інтервального повторення та система Лейтнера, є високоефективним, проте їх мануальне застосування супроводжується значними логістичними труднощами та потребує автоматизації.

Дослідження існуючих програмних альтернатив показало, що попри зручний функціонал для перевірки знань, ці платформи мають суттєвий недолік: відсутність зручних інтегрованих інструментів для масового автоматизованого управління персональними словниковими базами. Це обґрунтувало потребу у створенні спеціалізованого адміністративного модуля з функціями автоматизованого імпорту, гнучкого експорту та надійного розмежування прав доступу.

В ході виконання практичної частини роботи було досягнуто наступних результатів:

- Побудовано архітектуру та структуру даних: спроєктовано комплексну розподілену архітектуру застосунку, що гарантує високий рівень гнучкості, масштабованості та незалежності бізнес-логіки від зовнішніх факторів. Для надійного збереження складної ієрархічної структури інформації (словники, слова та їхні значення) було розроблено та налаштовано реляційну базу даних.
- Інтегровано підсистему безпеки: успішно впроваджено механізми електронної ідентифікації та централізованого управління доступом на основі ролей. Це дозволило чітко розмежувати повноваження: адміністраторам надано привілейований доступ до маніпуляцій з контентом, тоді як права звичайних користувачів обмежено безпечним пошуком та переглядом лексики.

- Реалізовано ключовий функціонал: розроблено базові інструменти для повноцінного управління словниковими базами. Для зручності користувачів впроваджено функціонал наскрізного глобального пошуку, алгоритми фільтрації слів, а також аналітичну панель для відстеження загальної статистики словників.
- Автоматизовано наповнення та експорт баз: для усунення рутинної ручної праці створено автоматизовані сервіси імпортування лексики із зовнішніх онлайн-ресурсів та локальних структурованих файлів. Додатково реалізовано механізм трансформації та експорту сформованих словників у стандартизовані формати для їх подальшого використання у спеціалізованих системах інтервального повторення.
- Налаштовано інфраструктуру та тестування: усі компоненти розробленої системи було успішно обгорнуто в ізольовані віртуальні середовища, що забезпечило повністю автоматизоване та стабільне розгортання програмного комплексу. Крім того, для перевірки коректності роботи ключових алгоритмів та запобігання помилкам було розроблено та впроваджено комплекс автоматизованих модульних тестів.
- Продемонстровано функціонал системи: практично підтверджено коректне функціонування головної сторінки, наскрізного пошуку, автоматизованого імпорту даних, а також експорту словників для сторонніх навчальних систем.

У підсумку, реалізовані рішення дозволили створити надійний, безпечний та зручний інструмент, який повністю задовольняє вимоги щодо автоматизації процесів формування та адміністрування цифрових навчальних словників.

ПЕРЕЛІК ПОСИЛАНЬ

1. Information Overload. The Decision Lab: веб-ресурс. URL: https://thedecisionlab.com/reference-guide/psychology/information-overload (дата звернення: 25.04.2026).

2. Forgetting Curve. The Decision Lab: веб-ресурс. URL: https://thedecisionlab.com/reference-guide/psychology/forgetting-curve (дата звернення: 25.04.2026).

3. What Is The Forgetting Curve (And How Do You Combat It)? eLearning Industry: веб-ресурс. URL: https://elearningindustry.com/forgetting-curve-combat (дата звернення: 25.04.2026).

4. CEFR Levels Explained: Your Comprehensive FAQ Guide. Text Inspector: веб-ресурс. URL: https://textinspector.com/cefr-levels-faq-guide/ (дата звернення: 25.04.2026).

5. How many words do you need to speak a language? BBC: веб-ресурс. URL: https://www.bbc.com/news/world-44569277 (дата звернення: 25.04.2026).

6. Leitner System for Flashcards. Mometrix: веб-ресурс. URL: https://www.mometrix.com/academy/leitner-study-method/ (дата звернення: 25.04.2026).

7. Spaced Repetition Explained: How It Works, Best Intervals, Leitner System + Tools (2026). Okti: веб-ресурс. URL: https://okti.app/en/blog/spaced-repetition-explained/ (дата звернення: 25.04.2026).

8. Software Development Technologies: Key Tools and Frameworks. Orion InfoSolutions: веб-ресурс. URL: https://www.orioninfosolutions.com/blog/software-development-technologies (дата звернення: 25.05.2026).

9. Introduction to Programming Languages. GeeksforGeeks: веб-ресурс. URL: https://www.geeksforgeeks.org/computer-science-fundamentals/introduction-to-programming-languages/ (дата звернення: 25.05.2026).

10. What is a Typed language ? GeeksforGeeks: веб-ресурс. URL: https://www.geeksforgeeks.org/javascript/what-is-a-typed-language/ (дата звернення: 25.05.2026).

11. Programming Paradigms – Paradigm Examples for Beginners. freeCodeCamp: веб-ресурс. URL: https://www.freecodecamp.org/news/an-introduction-to-programming-paradigms/#heading-procedural-programming (дата звернення: 25.05.2026).

12. Understanding The Different Types Of Databases & When To Use Them. Rivery: веб-ресурс. URL: https://rivery.io/data-learning-center/database-types-guide/ (дата звернення: 25.05.2026).

13. What is SQL (Structured Query Language)? AWS: веб-ресурс. URL: https://aws.amazon.com/what-is/sql/ (дата звернення: 25.05.2026).

14. What's the Difference Between Web Apps, Native Apps, and Hybrid Apps? AWS: веб-ресурс. URL: https://aws.amazon.com/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/#whats-the-difference-between-web-apps-native-apps-and-hybrid-apps--8vh3pa (дата звернення: 25.05.2026).

15. Reword: веб-ресурс. URL: https://reword.app/uk/en (дата звернення: 25.04.2026).

16. Quizlet Free Vs. Paid & Alternatives 2026. Learn.org: веб-ресурс. URL: https://learn.org/articles/quizlet-free-vs-paid (дата звернення: 25.04.2026).

17. Anki vs Quizlet vs Spaced Repetition Apps for Medical Students (2026). iatroX: веб-ресурс. URL: https://www.iatrox.com/blog/anki-vs-quizlet-vs-spaced-repetition-apps-medical-students-2026 (дата звернення: 25.04.2026).

18. Quizlet Subscriptions. Quizlet: веб-ресурс. URL: https://quizlet.com/upgrade?source=header_plus&redir=%2Flatest (дата звернення: 25.04.2026).

19. Remembering is easier with Anki. Anki: веб-ресурс. URL: https://apps.ankiweb.net/ (дата звернення: 25.04.2026).

20. What is Software Architecture? vFunction: веб-ресурс. URL: https://vfunction.com/blog/what-is-software-architecture/ (дата звернення: 27.05.2026).

21. Layered Architecture. DevIQ: веб-ресурс. URL: https://deviq.com/architecture/layered-architecture/ (дата звернення: 27.05.2026).

22. Clean Architecture. DevIQ: веб-ресурс. URL: https://deviq.com/architecture/clean-architecture/ (дата звернення: 27.05.2026).

23. Vertical Slice Architecture. Milan Jovanović: веб-ресурс. URL: https://www.milanjovanovic.tech/blog/vertical-slice-architecture (дата звернення: 27.05.2026).

24. Difference between System Architecture and Software Architecture. GeeksforGeeks: веб-ресурс. URL: https://www.geeksforgeeks.org/system-design/difference-between-system-architecture-and-software-architecture/ (дата звернення: 27.05.2026).

25. Monolithic Architecture. GeeksforGeeks: веб-ресурс. URL: https://www.geeksforgeeks.org/system-design/monolithic-architecture-system-design/ (дата звернення: 27.05.2026).

26. What is a distributed system? Atlassian: веб-ресурс. URL: https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture (дата звернення: 27.05.2026).

27. Microservices. GeeksforGeeks: веб-ресурс. URL: https://www.geeksforgeeks.org/system-design/microservices/ (дата звернення: 27.05.2026).

28. C#. Microsoft: веб-ресурс. URL: https://dotnet.microsoft.com/en-us/languages/csharp

29. What is Object-Relational Mapping (ORM)? AWS: веб-ресурс. URL: https://aws.amazon.com/what-is/object-relational-mapping/ (дата звернення: 27.05.2026).

30. Language Integrated Query (LINQ). Microsoft: веб-ресурс. URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq/ (дата звернення: 27.05.2026).

31. Dapper Vs Entity Framework Core. C# Corner: веб-ресурс. URL: https://www.c-sharpcorner.com/article/dapper-vs-entity-framework-core/ (дата звернення: 27.05.2026).

32. Overview of ASP.NET Core. Microsoft: веб-ресурс. URL: https://learn.microsoft.com/en-us/aspnet/core/overview?view=aspnetcore-10.0 (дата звернення: 27.05.2026).

33. ASP.NET Core Blazor. Microsoft: веб-ресурс. URL: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-10.0 (дата звернення: 27.05.2026).

34. WebAssembly. WebAssembly: веб-ресурс. URL: https://webassembly.org/ (дата звернення: 27.05.2026).

35. About. PostgreSQL: веб-ресурс. URL: https://www.postgresql.org/about/ (дата звернення: 27.05.2026).

36. What is Docker? IBM: веб-ресурс. URL: https://www.ibm.com/think/topics/docker (дата звернення: 27.05.2026).

37. Open Source Identity and Access Management. Keycloak: веб-ресурс. URL: https://www.keycloak.org/ (дата звернення: 27.05.2026).

38. Entity Framework Core – Code First Approach. C# Corner: веб-ресурс. URL: https://www.c-sharpcorner.com/article/entity-framework-core-code-first-approach/ (дата звернення: 27.05.2026).

39. Creating and Configuring a Model. Microsoft: веб-ресурс. URL: https://learn.microsoft.com/en-us/ef/core/modeling/ (дата звернення: 27.05.2026).

40. Authentication vs. Authorization. Auth0: веб-ресурс. URL: https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization (дата звернення: 27.05.2026).

41. Role-Based Access Control. Auth0: веб-ресурс. URL: https://auth0.com/docs/manage-users/access-control/rbac (дата звернення: 27.05.2026).

42. Unit Testing. GeeksForGeeks: веб-ресурс. URL: https://www.geeksforgeeks.org/software-testing/unit-testing-software-testing/ (дата звернення: 27.05.2026).