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

ГАПОЯНЦ ДАРИНА ВОЛОДИМИРІВНА

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

РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ДОДАТКУ ДЛЯ ЧИТАННЯ ІЗ ВБУДОВАНИМ ІНТЕРАКТИВНИМ СЛОВНИКОМ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Ірина ФРИЗ, старший викладач
кафедри інформаційних технологій,
канд. фіз.-мат. наук
_____

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

Голова ЕК: _____
(підпис)

Вінниця – 2026

АНОТАЦІЯ

Гапоянц Д.В. Розробка серверної частини мобільного додатку для читання із вбудованим інтерактивним словником. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2026.

У кваліфікаційній (бакалаврській) роботі розглянуто проєктування та реалізацію серверної частини мобільного застосунку для читання книг із підтримкою інтерактивного словника та навчальних механізмів для повторення лексики. У роботі проаналізовано сучасні підходи до побудови REST API для мобільних застосунків, методи організації серверної архітектури, засоби автентифікації користувачів і підходи до взаємодії із зовнішніми сервісами.

Ключові слова: серверна частина, мобільний додаток, REST API, FastAPI, MySQL, SQLAlchemy ORM, JWT-автентифікація, інтерактивний словник, Flutter, Translation API.

105 ст., 14 рис., 3 табл., 2 дод., 57 джерел.

ABSTRACT

Hapoyants D. Development of the Server Part of a Mobile Reading Application with a Built-In Interactive Dictionary. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl' Stus Donetsk National University, Vinnytsia, 2026.

The qualification (bachelor's) thesis focuses on the design and implementation of the server-side architecture of a mobile reading application with an integrated interactive dictionary and vocabulary learning functionality. The study examines modern approaches to REST API development for mobile systems, backend architecture organization, utаблицяser authentication mechanisms, and integration with external services.

Keywords: server-side application, mobile application, REST API, FastAPI, MySQL, SQLAlchemy ORM, JWT authentication, interactive dictionary, Flutter, Translation API.

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій цифрові сервіси для навчання та самоосвіти набувають дедалі більшого значення. Зокрема, особливу популярність отримують додатки для читання текстів іноземними мовами, які поєднують у собі функції доступу до літератури та інструменти для покращення мовних навичок. Однією з ключових проблем, з якою стикаються користувачі таких додатків, є необхідність постійного звернення до сторонніх словників, що ускладнює процес читання та знижує його ефективність.

Актуальність теми полягає у потребі створення ефективних програмних рішень, які забезпечують інтеграцію інтерактивного словника безпосередньо в процес читання. Це дозволяє користувачам швидко отримувати переклад та пояснення незнайомих слів без переривання взаємодії з текстом. Важливу роль у реалізації таких функцій відіграє серверна частина додатку, яка відповідає за обробку запитів, збереження даних, взаємодію з базами даних та забезпечення стабільної роботи системи.

Метою даної бакалаврської роботи є розробка серверної частини додатку для читання з вбудованим інтерактивним словником, що забезпечує ефективну обробку текстових даних, швидкий доступ до словникових ресурсів та зручну взаємодію користувача з системою.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати існуючі рішення у сфері додатків для читання та інтерактивних словників;
- визначити функціональні та нефункціональні вимоги до серверної частини системи;
- обрати технології та інструменти для реалізації серверної архітектури;
- розробити структуру бази даних;
- реалізувати основні функції серверної частини додатку;
- провести тестування та оцінку ефективності розробленого рішення.

Об’єктом дослідження є процес організації серверної взаємодії у веб- або мобільних додатках для читання.

Предметом дослідження є методи та засоби розробки серверної частини додатку з інтегрованим інтерактивним словником.

Практичне значення отриманих результатів полягає у можливості використання розробленого програмного забезпечення для створення сучасних освітніх платформ, що сприяють ефективному вивченню іноземних мов та покращенню навичок читання.

Апробація результатів дослідження. Результати дослідження доповідалися на VII Всеукраїнській науково-практичній конференції здобувачів вищої освіти та молодих вчених «Прикладні інформаційні технології», 15 травня 2026 р., Донецький національний університет імені Василя Стуса, м. Вінниця, тема доповіді: «Архітектурні особливості серверної частини мобільного застосунку для читання з інтерактивним словником».

Структура та обсяг роботи. Бакалаврська робота складається зі вступу, трьох основних розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 56 сторінок

У **вступі** обґрунтовано актуальність теми, визначено мету, завдання, об’єкт та предмет дослідження, а також наукову новизну та практичне значення отриманих результатів.

У **першому розділі** «Аналіз предметної області та технологій додатків для читання», проаналізовано теоретичні засади розвитку таких систем, розглянуто принципи роботи інтерактивних словників, проведено порівняльний аналіз наявних аналогів та обґрунтовано вибір технологічного стеку.

У **другому розділі** «Проєктування та реалізація серверної частини системи», описано архітектуру системи, спроєктовано базу даних, а також визначено принципи побудови REST API та механізмів безпеки.

У **третьому розділі** «Проєктування та реалізація серверної частини системи», описано архітектуру системи, спроєктовано базу даних, а також визначено принципи побудови REST API та механізмів безпеки.

У **висновках** узагальнено результати дослідження, підтверджено досягнення поставленої мети та визначено перспективи подальшого розвитку проєкту.Додатки містять лістинги ORM-моделей та фрагменти програмного коду, що ілюструють реалізацію окремих архітектурних рішень.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕХНОЛОГІЙ ДОДАТКІВ ДЛЯ ЧИТАННЯ

1.1 Теоретичні засади розвитку додатків для читання та цифрових платформ

У сучасному світі технології займають невід'ємну частку повсякденного життя кожної людини. Цифровізація супроводжує нас усюди: від відправлення робочих емейлів до перегляду стрічки в інстаграмі. Лише на мобільні телефони в середньому люди витрачають 4,8 годин на день, що становить близько 30% їхнього часу [1]. У телефоні людина зосереджує майже увесь свій вільний час, переглядаючи відео, читаючи новинну стрічку чи просто комунікуючи з близькими. Не винятком є і читання книг, що на сьогоднішній день, попри появу цифрових технологій, не втратило свою популярність. Щоправда, в конкуренції з соціальними мережами та стримінговими платформами, класична паперова книга видозмінила свій традиційний вигляд і трансформувалась у щось, чого стародавня людина не могла собі навіть уявити – електронну книгу. Тепер можна читати будь-де і коли-завгодно, незважаючи на обставини, адже новий формат книги важить надзвичайно легко і майже завжди супроводжує людину.

Під електронною книгою (e-book) у межах даної роботи ми розуміємо цифровий формат книжкового видання, що містить текстову та графічну інформацію, адаптовану для відображення на мобільних пристроях, і забезпечує можливість інтерактивної взаємодії з контентом.

Опитування показують, що хоч і друковані книги залишаються улюбленим форматом більшості опитуваних, популярність споживання електронних книг у світі стрімко зростає. Зокрема, станом на 2023 рік 33% українців віддають перевагу читанню електронної книги, що на 6% більше ніж у 2020 [2].

Переваги електронних книг над паперовими [3]:

1. Можливість переносити велику кількість книг, яка не буде займати додаткове місце у валізі чи рюкзаку і спричиняти зайвий клопіт.

2. Для електронної книги не потрібно шукати додаткове джерело освітлення, на відміну від паперової.

3. Паперові книги є менш екологічними і можуть легко пошкодитись.

4. Вбудовані словники та можливість кастомізації електронної книги.

5. Легкий процес публікації у порівнянні видавництвом друкованої книги.

6. Легка доступність електронних книг в інтернет ресурсах.

7. Порівняно невелика вартість електронного формату над паперовим.

Зростання популярності електронних книг спричиняє стрімкий розвиток та пошук нових рішень у сфері цифрових платформ для читання [4]. Особливого значення такі додатки набувають у контексті вивчення іноземних мов [5]. Читання оригінальних текстів є ефективним способом розширення словникового запасу, проте воно часто ускладнюється наявністю незнайомої лексики. Саме тому виникає потреба у впровадженні додаткових інструментів, які дозволяють швидко отримувати значення слів без втрати контексту [6]. Це призвело до появи інтерактивних функцій, зокрема вбудованих словників, які стали невід'ємною частиною сучасних додатків для читання [7].

Таким чином, розвиток додатків для читання відбувається у напрямку підвищення їхньої функціональності, інтерактивності та орієнтації на потреби користувача, що створює передумови для розробки більш складних програмних систем із використанням серверних технологій.

1.2 Сутність та принципи роботи інтерактивних словників у процесі читання

Інтерактивний словник є важливим компонентом сучасних систем для читання, особливо у випадках, коли мова йде про тексти іноземною мовою [8]. Його основне призначення полягає у забезпеченні швидкого доступу до перекладу або тлумачення слів без необхідності залишати середовище читання.

Це значно підвищує зручність використання додатку та ефективність засвоєння інформації.

Принцип роботи інтерактивного словника базується на взаємодії клієнтської та серверної частин системи. Користувач, взаємодіючи з текстом, обирає певне слово або фрагмент, після чого формується запит, який передається на сервер. Серверна частина обробляє цей запит, звертається до бази даних або зовнішніх сервісів і повертає відповідь у вигляді перекладу, визначення або додаткової інформації про слово. Отримані дані відображаються користувачу у зручному форматі, наприклад у вигляді спливаючого вікна.

Важливою особливістю інтерактивних словників є їх здатність працювати з урахуванням контексту, що дозволяє підвищити точність перекладу. Крім того, такі системи можуть зберігати історію переглянутих слів, формувати персональний словник користувача та адаптувати результати відповідно до його рівня знань [9].

Інтеграція словника у додаток може реалізовуватися різними способами:

- Локальний: передбачає збереження бази даних на пристрої (наприклад, SQLite). Забезпечує автономність, проте обмежує можливості актуалізації даних та збільшує розмір застосунку.
- Гібридний: поєднує локальне кешування найвживанішої лексики з динамічними запитами до API. Потребує складних механізмів синхронізації стану.
- Серверний (API-based): система взаємодіє з віддаленим сервером для обробки запитів. Цей підхід забезпечує масштабованість, можливість централізованого оновлення словникового контенту та інтеграції із зовнішніми Translation API без внесення змін до клієнтської частини.

У межах реалізованої системи обрано серверний підхід, оскільки він є найбільш доцільним для архітектури, що потребує гнучкого розширення навчального функціоналу та ефективної обробки даних користувача.

1.3 Аналіз існуючих програмних рішень для читання з інтегрованими словниками

На ринку вже є численні продукти для читання з вбудованими словниками. Так, сервіс Amazon Kindle пропонує великий каталог електронних видань і вбудований словник, але при цьому залишається частково закритим середовищем (потрібна реєстрація та купівля через Amazon). У ньому є базові функції: вибір шрифту, закладки, підсвічування, синтезатор мови, але бракує гнучких інструментів для навчання (наприклад, флеш-картки з вивченими словами). Аналогічно Google Play Books дозволяє завантажувати власний контент і має перекладач Google, однак теж не фокусований на систематичному запам’ятовуванні лексики. Існують також спеціалізовані мовні додатки – наприклад, LingQ чи Readlang – які здебільшого орієнтовані на вивчення іноземних слів із контексту [10]. Вони забезпечують інтерактивний переклад і ведення словника користувача, але не завжди підходять для повсякденного читання довільного тексту та можуть вимагати платного доступу.

Внаслідок цього можна виокремити декілька переваг і недоліків наявних аналогів. Загальними перевагами є: доступ до великих бібліотек контенту, синхронізація між пристроями, простота читання та базова функціональність (пошук, анотації). Недоліками – відсутність налаштування під конкретного користувача, обмеженість у підтримці нових форматів чи індивідуальних словників, а також брак глибоких навчальних опцій. Наприклад, лише 4% традиційних читачів долучаються до великих обсягів (понад 50 книг), у той час як серед користувачів електронних носіїв таких удвічі більше – і існуючі додатки не завжди стимулюють такий розвиток [11]. Результати порівняльного аналізу функціональних можливостей популярних додатків для читання та розроблюваної системи систематизовано в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз функціональних можливостей аналогів

Критерій порівняння	**Amazon Kindle**	**Google Play Books**	**LingQ / Readlang**	**Розроблювана система**
Робота з власним контентом	Обмежена	Висока	Середня	Висока
Інтерактивний переклад	Базовий	Високий (Google)	Високий	Високий
Навчальний функціонал (flashcards)	Відсутній	Відсутній	Наявний	Наявний
Модель доступу	Екосистемна (закрита)	Відкрита	Переважно платна	Безкоштовна
Ергономіка читання	Висока	Висока	Середня	Висока
Систематизація лексики	Низька	Низька	Висока	Висока

Як свідчать дані таблиці 1.1, розроблювана система ефективно інтегрує переваги обох сегментів ринку: забезпечує високу ергономіку читання, характерну для класичних рідерів, та просунутий інструментарій систематизації лексики, притаманний спеціалізованим лінгвістичним платформам. Такий синтез дозволяє користувачеві опановувати іноземну мову безпосередньо під час читання довільного контенту, усуваючи функціональні обмеження наявних аналогів.

На основі здійсненого аналізу, до нової системи пред’являються такі функціональні вимоги:

- надання єдиної платформи для читання різноформатних електронних книжок (серверна бібліотека з можливістю додавання користувацьких творів);
- інтеграція інтерактивного словника, що миттєво показує тлумачення слів і дозволяє додавати невідомі слова до особистого словника;
- механізми флеш-карток та тестів для повторення вивченої лексики на основі прочитаних текстів;
- гнучкі налаштування інтерфейсу: різні теми, шрифти, регулювання яскравості, безшовний пошук та навігація текстом.

Ці вимоги закладають основу подальшого проєктування системи та виявляють прогалини в існуючих програмних рішеннях. Підходи до реалізації і реалізацію визначених функціональних вимог опишемо у наступних розділах

1.4 Обґрунтування вибору технологій для реалізації системи

Розробка серверної частини мобільного додатку для читання з інтерактивним словником потребує використання технологій, які підтримують швидке створення REST API, інтеграцію з мобільним клієнтом, роботу з реляційною базою даних та взаємодію із зовнішніми сервісами [12]. Вибір технологічного стеку у межах проєкту здійснювався з урахуванням вимог до модульності системи, підтримуваності backend-коду, швидкості розробки та сумісності між окремими компонентами архітектури.

Основною мовою програмування серверної частини обрано Python. Це рішення пов’язане з широкою підтримкою backend-розробки, наявністю сучасних фреймворків для створення API та великою кількістю бібліотек для інтеграції із зовнішніми сервісами. Python добре підходить для реалізації REST-oriented backend-систем середнього масштабу, де важливими є швидкість розробки та читабельність коду. Порівняно з мовами на кшталт Java або C#, Python має нижчий поріг входу та спрощує прототипування серверної логіки [13]. Порівняння основних мов програмування для серверної частини системи наведено у таблиці 1.2.

Таблиця 1.2 – Порівняльний аналіз технологічного стеку для backend-розробки

Критерій порівняння	**Python (FastAPI)**	**Java (Spring Boot)**	**C# (ASP.NET Core)**
Поріг входу	Низький	Високий	Середній
Швидкість розробки	Висока	Середня	Середня
Прототипування	Швидке	Складне	Помірне
Продуктивність	Висока (Async)	Дуже висока	Дуже висока
Читабельність коду	Висока	Середня	Середня
Екосистема API	Спеціалізована	Корпоративна	Корпоративна

Як свідчать дані таблиці 1.2, Python із фреймворком FastAPI є найбільш доцільним вибором для даного проєкту. Основні аргументи на користь цього рішення:

- Швидкість прототипування: завдяки синтаксичній лаконічності та підтримці сучасних парадигм (асинхронність, типи Pydantic), Python дозволяє реалізувати REST API значно швидше за Java чи C#.
- Оптимізація для API: на відміну від громіздких корпоративних фреймворків, FastAPI спроєктований спеціально для створення легких, високопродуктивних систем, що працюють з JSON, що ідеально відповідає архітектурі мобільного додатку.
- Низький поріг входу: це забезпечує легкість підтримки та можливість внесення змін у бізнес-логіку без необхідності роботи з надлишковим boilerplate-кодом.

Попри вищу продуктивність Java або C# у високонавантажених корпоративних системах, Python забезпечує оптимальний баланс між швидкістю розробки, читабельністю коду та якістю інтеграції з мобільним клієнтом у межах бакалаврського проєкту.

Для реалізації backend-рівня використовується FastAPI. Вибір цього framework обумовлений підтримкою асинхронної обробки HTTP-запитів, інтеграцією з Pydantic та вбудованою OpenAPI-документацією. FastAPI добре підходить для mobile-oriented REST API, оскільки спрощує створення endpoint-ів і підтримує dependency injection через Depends() [14].

Порівняно з Django REST Framework FastAPI має менш громіздку структуру та краще підходить для lightweight backend-систем, де API є центральною частиною архітектури. Крім того, FastAPI підтримує асинхронний execution model, що є корисним під час інтеграції із зовнішнім Translation API та виконання I/O-bound операцій.

Разом із цим екосистема FastAPI є менш зрілою порівняно з Django. Django REST Framework має ширший набір готових адміністративних компонентів і вбудованих security-механізмів, однак для поточного проєкту використання FastAPI є більш доцільним через орієнтацію системи саме на REST API та взаємодію з мобільним клієнтом [15].

Клієнтська частина реалізована за допомогою Flutter. Основною причиною вибору Flutter є можливість створення cross-platform mobile application на основі єдиної codebase. Для системи, у якій frontend виконує роль інтерфейсу взаємодії з REST API, такий підхід спрощує інтеграцію між клієнтською і серверною частинами [16].

Для зберігання даних використовується MySQL. Вибір реляційної СУБД пов’язаний зі структурованим характером даних системи: користувачі, книги, закладки, словникові записи та результати quiz-модуля мають чіткі зв’язки між собою. Використання relational database спрощує підтримку цілісності даних та забезпечує узгодженість між сутностями [17].

Порівняно з NoSQL databases MySQL краще підходить для систем із великою кількістю взаємопов’язаних сутностей та необхідністю виконання JOIN-операцій. NoSQL-рішення є більш гнучкими у сценаріях роботи з неструктурованими даними, однак у межах поточного проєкту relational model є більш природною для реалізації зв’язків між ORM-моделями.

Для взаємодії backend із базою даних використовується SQLAlchemy ORM. ORM-підхід дозволяє працювати з даними через Python-об’єкти замість ручного написання SQL-запитів. Це підвищує читабельність backend-коду та спрощує підтримку зв’язків між сутностями. Приклади ORM-моделей та зв’язків серверної частини наведено у додатку Б.

Порівняно з raw SQL SQLAlchemy спрощує реалізацію CRUD-операцій, ownership validation та роботу зі зв’язками. Крім того, ORM-рівень краще інтегрується з FastAPI та архітектури ін’єкції залежностей. Водночас ORM створює додатковий рівень абстракції між застосунком і базою даних, що може ускладнювати оптимізацію складних SQL-запитів у високонавантажених сценаріях.

Для валідації HTTP-запитів та відповідей використовується Pydantic. Основною перевагою Pydantic є автоматична перевірка типів даних та інтеграція з FastAPI. Backend-система отримує додатковий рівень контролю над request/response structures, що спрощує підтримку узгодженості між API та ORM-моделями.

Порівняно з ручною валідацією даних використання Pydantic зменшує кількість boilerplate-коду та централізує validation logic. Разом із цим складні nested DTO можуть ускладнювати структуру схем при збільшенні кількості API-моделей.

Для реалізації механізмів автентифікації обрано JWT. Такий підхід добре інтегрується зі stateless mobile API та дозволяє Flutter-клієнту працювати із захищеними endpoint-ами без використання server-side session storage [18].

Порівняно з session-based authentication JWT краще підходить для mobile-oriented architectures, де клієнт взаємодіє із сервером через Bearer token. Крім того, JWT спрощує масштабування backend-системи, оскільки сервер не зберігає інформацію про активні сесії користувачів.

Разом із цим JWT має певні обмеження. Stateless nature токенів ускладнює реалізацію token revocation та logout-mechanism. Якщо токен був

скомпрометований, він залишатиметься валідним до завершення строку дії, якщо не реалізовано додаткову blacklist architecture.

Для хешування паролів використовується bcrypt. Зберігання password hash замість plaintext password є базовою вимогою до безпеки backend-систем. bcrypt підтримує adaptive hashing, що ускладнює brute-force атаки та підвищує стійкість системи до компрометації бази даних [19].

Для інтеграції із зовнішнім Translation API використовується бібліотека httpx. Її вибір пов'язаний із підтримкою асинхронних HTTP-запитів та сумісністю з FastAPI asynchronous flow. Backend надсилає запити до зовнішнього сервісу перекладу та повертає клієнту уніфікований JSON-результат [20].

Порівняно з бібліотекою requests, httpx краще інтегрується з asynchronous architecture FastAPI. Це особливо важливо для I/O-bound сценаріїв, де сервер очікує відповідь зовнішнього API. Разом із цим використання зовнішнього Translation API створює залежність від стороннього сервісу, його стабільності та якості перекладу.

Для локальної інтеграції Flutter-клієнта та FastAPI backend використовувався ngrok. Сервіс дозволяє створити публічний HTTPS endpoint для локального сервера без необхідності розгортання production infrastructure. Це суттєво спрощує тестування мобільного застосунку на фізичних пристроях.

Разом із цим ngrok не є production-oriented рішенням. Безкоштовна версія сервісу має тимчасові URL-адреси та обмеження на тривалість сесій. Для production deployment подібний підхід не є достатнім через питання стабільності та безпеки.

Технологічний стек проєкту сформовано з урахуванням специфіки mobile-oriented backend-системи. FastAPI, SQLAlchemy ORM, MySQL та JWT добре інтегруються між собою та підтримують реалізацію REST API для мобільного застосунку. Flutter забезпечує кросплатформену клієнтську частину, а інтеграція із зовнішнім Translation API дозволяє реалізувати інтерактивний словниковий функціонал без створення власної системи машинного перекладу.

Обрані технології формують достатню основу для реалізації функціоналу бакалаврського проєкту та підтримують подальший розвиток системи. Разом із цим окремі архітектурні рішення, зокрема JWT authentication і залежність від зовнішнього Translation API, створюють характерні архітектурні компроміси, які необхідно враховувати під час подальшого масштабування backend-системи.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ

2.1 Проєктування архітектури системи

Проєктування архітектури серверної частини є одним із ключових етапів розробки мобільного додатку для читання з інтерактивним словником, оскільки саме на цьому рівні визначаються принципи взаємодії між компонентами системи, механізми обробки даних та організація серверної логіки. Архітектурні рішення безпосередньо впливають на масштабованість, підтримуваність та можливість подальшого розширення функціоналу системи.

Для реалізації системи було обрано клієнт-серверну архітектуру, у межах якої Flutter-клієнт відповідає за користувацький інтерфейс і взаємодію з користувачем, а FastAPI-сервер виконує обробку запитів, роботу з базою даних, автентифікацію та інтеграцію із зовнішнім сервісом перекладу. Подібний підхід забезпечує незалежність клієнтської та серверної частин, що спрощує супровід системи та дозволяє модифікувати клієнтську частину або серверну частину без необхідності повного перепроєктування застосунку. [21]

Архітектура системи побудована навколо REST API, через який Flutter-клієнт взаємодіє із серверною частиною. Використання REST-підходу є доцільним для мобільних застосунків, оскільки HTTP-запити та JSON-відповіді підтримуються більшістю сучасних платформ і забезпечують стандартизований спосіб передачі даних між компонентами системи. Ресурсно-орієнтована архітектура дозволяє організувати API навколо окремих сутностей системи, зокрема книг, словникових записів, закладок та навчальних модулів. [22]

REST-архітектура також відповідає принципу безстанового обміну даними. Сервер не зберігає інформацію про стан клієнтської сесії між запитами, а кожен HTTP-запит містить усі необхідні дані для його обробки. Подібний підхід спрощує масштабування backend-частини та зменшує зв’язність між окремими запитами. Разом із цим безстанова архітектура підвищує вимоги до

механізмів авторизації, оскільки перевірка доступу виконується для кожного запиту окремо. [23]

У ролі серверного фреймворку використовується FastAPI. Структура БД побудована відповідно до принципів третьої нормальної форми (3НФ), що дозволяє усунути надлишковість даних та спростити їх оновлення. Асинхронна модель роботи FastAPI є актуальною для серверних систем, що взаємодіють із зовнішніми API або виконують одночасну обробку великої кількості HTTP-запитів. У контексті розробленої системи це важливо для модуля перекладу, який виконує мережеві запити до зовнішнього Translation API. [24]

Для візуалізації взаємодії користувача із системою було побудовано Use Case Diagram, яка демонструє основні сценарії роботи мобільного додатку: авторизацію користувача, читання книг, збереження прогресу, створення закладок, переклад слів, ведення персонального словника та проходження навчальних quiz-сценаріїв (рис. 2.1).

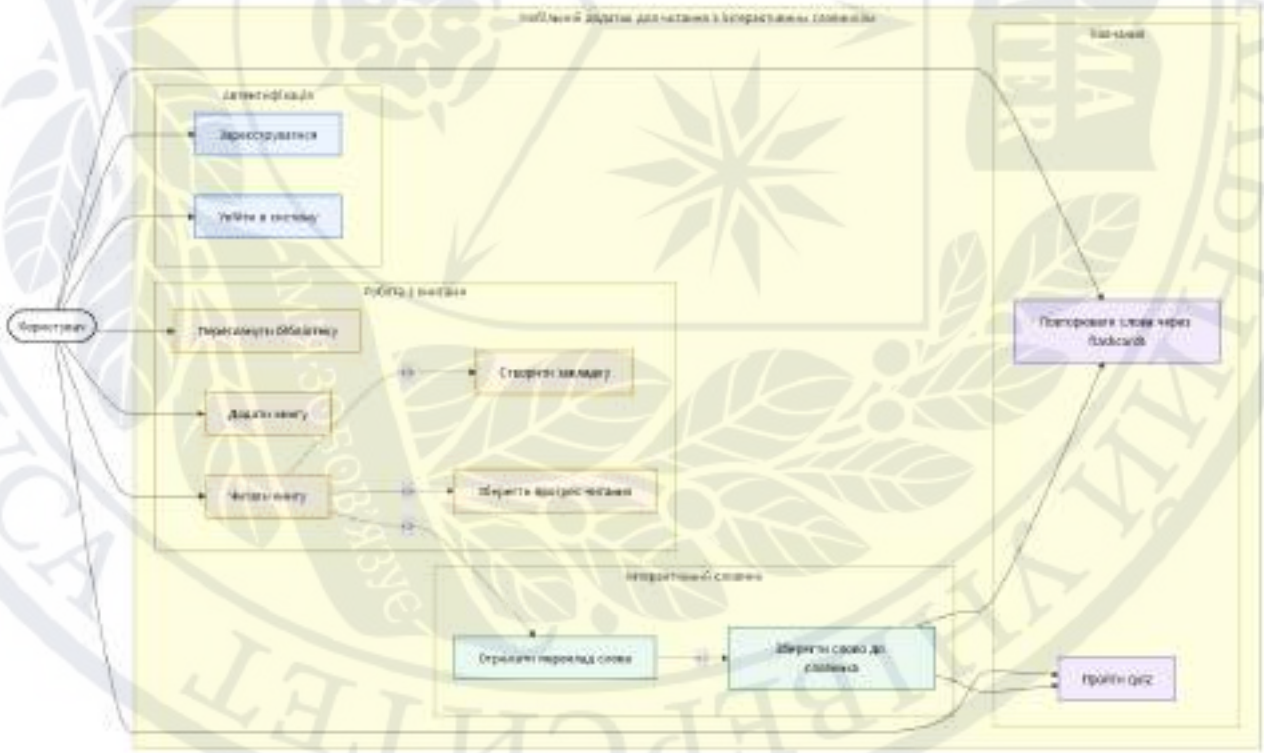

Рисунок 2.1 – Use Case Diagram мобільного додатку для читання з інтерактивним словником

Важливою особливістю архітектури є модульна організація backend-частини. Серверна логіка поділена на окремі router-модулі відповідно до функціонального призначення: авторизація, книги, прогрес читання, закладки, словник, flashcards та quiz. Подібний підхід відповідає принципу розділення відповідальності, згідно з яким кожен компонент системи відповідає лише за визначену частину бізнес-логіки. [25] Фрагменти коду ініціалізації FastAPI-застосунку та підключення router-модулів наведено у додатку Б.

Модульна архітектура позитивно впливає на підтримуваність системи. Ізоляція функціональних компонентів спрощує внесення змін до окремих частин застосунку та зменшує ризик порушення роботи інших модулів. Наприклад, модифікація алгоритму інтервального повторення для flashcards не потребує змін у модулі авторизації або логіці роботи з книгами. Подібна структура також спрощує масштабування backend-системи, оскільки окремі компоненти можуть бути розширені або винесені в окремі сервіси без суттєвого перепроєктування всієї архітектури.

Для організації взаємодії між компонентами backend використовується ін’єкція залежностей, реалізований через механізм Depends у FastAPI. Ін’єкція залежностей застосовується для централізованого підключення сесія бази даних, механізмів авторизації та інших спільних компонентів системи. Подібний підхід зменшує дублювання коду між endpoint-ами та робить backend більш підтримуваним.

Наприклад, dependency get_db відповідає за створення та завершення database session для кожного HTTP-запиту, тоді як get_current_user використовується для централізованої перевірки JWT-токена та отримання інформації про поточного користувача. Така організація backend-логіки спрощує реалізацію контроль авторизації та підтримує узгодженість між окремими модулями системи. Приклади реалізації dependency injection через get_db() та get_current_user наведено у додатку Б.

Для роботи з базою даних використовується SQLAlchemy ORM. ORM-підхід дозволяє працювати із записами бази даних у вигляді Python-об’єктів, що

спрощує реалізацію зв’язків між сутностями та робить backend-код більш читабельним. Порівняно з використанням raw SQL, ORM забезпечує вищий рівень абстракції та зменшує кількість повторюваного коду. Разом із цим ORM може створювати додаткові накладні витрати у високонавантажених системах або під час виконання складних SQL-запитів, однак для реалізації функціоналу бакалаврського проєкту достатньо можливостей SQLAlchemy [26]. Приклади ORM-моделей та зв’язків серверної частини наведено у додатку Б.

У ролі системи керування базами даних використовується MySQL. Вибір MySQL обумовлений підтримкою реляційної моделі даних, ForeignKey-залежностей, indexes та механізмів забезпечення цілісності даних. Реляційний підхід є доцільним для системи, у якій між сутностями існує велика кількість структурованих зв’язків: користувачі пов’язані з книгами, книги − із закладками та прогресом читання, а словникові записи − із навчальними модулями [27, 28].

Для інтеграції функціоналу перекладу використовується зовнішній Translation API. Backend виконує роль проміжного рівня між Flutter-клієнтом та зовнішнім сервісом перекладу: сервер приймає запит від клієнта, надсилає його до Translation API та повертає уніфіковану відповідь у форматі JSON. Подібний підхід дозволяє реалізувати live-переклад без необхідності створення власної системи машинного перекладу. Приклади інтеграції із зовнішнім Translation API та реалізації endpoint-а перекладу наведено у додатку Б.

Окрему увагу під час проєктування архітектури було приділено механізмам безпеки. Для автентифікації користувачів використовується JWT-підхід, у межах якого сервер генерує токен доступу після успішного входу користувача в систему. Stateless JWT-автентифікація добре інтегрується з REST API та спрощує взаємодію між Flutter-клієнтом і серверною частиною. Разом із перевагами подібний підхід має певні обмеження, зокрема складнішу реалізацію logout та механізмів відкликання токенів.

Для ізоляції конфігураційних параметрів застосунку використовується файл .env. У ньому зберігаються параметри підключення до бази даних, секретний ключ JWT, адреса Translation API та інші конфігураційні значення.

Використання змінних середовища спрощує перенесення застосунку між різними середовищами розробки та зменшує ризик випадкового розміщення конфіденційних даних безпосередньо у вихідному коді [29].

Таким чином, спроєктована архітектура поєднує REST API, модульну організацію backend-логіки, ORM-підхід до роботи з базою даних та JWT-автентифікацію. Подібна структура є достатньою для реалізації функціональних вимог мобільного додатку та створює основу для подальшого розвитку системи. Фрагменти ініціалізації FastAPI-застосунку та підключення router-модулів наведено у додатку Б.

2.2 Проєктування бази даних системи

Проєктування структури бази даних є одним із ключових етапів розробки серверної частини мобільного додатку, оскільки саме база даних визначає спосіб зберігання інформації, підтримку зв’язків між сутностями та цілісність даних системи. Для реалізації backend-частини було обрано реляційну модель даних на основі MySQL, що є доцільним для систем із великою кількістю структурованих взаємозалежних сутностей [30]. У контексті розробленого застосунку такими сутностями є користувачі, книги, прогрес читання, закладки, словникові записи та результати навчальних модулів.

Під час проєктування бази даних основний акцент було зроблено на забезпеченні нормалізації, мінімізації дублювання інформації та підтримці цілісності даних між таблицями. Структура БД побудована відповідно до принципів третьої нормальної форми (3НФ), що дозволяє усунути надлишковість даних та спростити їх оновлення. Кожна таблиця відповідає окремій логічній сутності системи, а зв’язки між сутностями реалізуються через механізм ForeignKey [31].

Основу системи становить таблиця users, яка містить інформацію про зареєстрованих користувачів. Для кожного користувача зберігаються унікальний ідентифікатор, адреса електронної пошти, ім’я користувача, хеш пароля та дата

створення облікового запису. Поля email і username мають UNIQUE constraints, що унеможливлює створення декількох облікових записів із однаковими даними. Подібне рішення підтримує цілісність системи автентифікації та спрощує механізм пошуку користувачів.

Таблиця books призначена для зберігання інформації про книги, додані користувачем. Кожен запис містить назву книги, автора, опис, текстовий вміст та мовні параметри. Між таблицями users і books реалізовано зв'язок one-to-many: один користувач може мати декілька книг, тоді як кожна книга належить лише одному користувачеві. Для цього використовується зовнішній ключ user_id, який посилається на первинний ключ таблиці users.

З метою підтримки цілісності даних для зв'язку між users і books використовується ON DELETE CASCADE. У разі видалення користувача автоматично видаляються пов'язані книги, що дозволяє уникнути появи orphan records − записів, які втратили зв'язок із батьківською сутністю. Подібний підхід спрощує адміністрування даних, однак потребує обережного використання, оскільки каскадне видалення може призвести до втрати значної кількості пов'язаної інформації [32].

Для підтримки функціоналу продовження читання використовується таблиця reading_progress. Вона зберігає поточну позицію користувача у тексті книги та дозволяє автоматично відновлювати місце читання після повторного відкриття книги. Таблиця реалізує зв'язки many-to-one із таблицями users та books через зовнішні ключі user_id і book_id.

Оскільки для кожної пари «користувач–книга» має існувати лише один запис прогресу читання, у таблиці reading_progress використовується UNIQUE constraint для комбінації user_id і book_id. Це унеможливлює дублювання записів та спрощує оновлення поточної позиції читання.

Система закладок реалізована через таблицю bookmarks, яка містить інформацію про вибрані фрагменти тексту, позицію в книзі та додаткові нотатки користувача. Між таблицями books і bookmarks реалізовано зв'язок one-to-many: одна книга може містити багато закладок. Аналогічно до інших залежних

сутностей, для bookmarks використовується ON DELETE CASCADE, що гарантує автоматичне видалення закладок після видалення книги.

Окрему роль у структурі БД відіграє таблиця dictionary_entries. Вона містить унікальні словникові записи, які використовуються в інтерактивному словнику та навчальних модулях. Винесення словникових записів в окрему таблицю дозволяє уникнути дублювання однакових перекладів для різних користувачів. Замість збереження одного й того самого слова декілька разів система використовує централізовану колекцію перекладених термінів.

У таблиці dictionary_entries застосовано декілька UNIQUE constraints, що запобігають дублюванню словникових записів. Наприклад, комбінація word, source_language та target_language повинна бути унікальною. Подібне рішення зменшує обсяг дубльованих даних у БД та спрощує подальшу роботу навчальних модулів.

Для реалізації персонального словника користувача використовується таблиця saved_words, яка виступає проміжною сутністю між users та dictionary_entries. Вона реалізує зв’язок many-to-many: один користувач може зберігати багато слів, а один словниковий запис може використовуватися багатьма користувачами.

Крім зв’язків із користувачами та словниковими записами, таблиця saved_words містить поля, необхідні для реалізації flashcards і інтервального повторення. До них належать difficulty, next_review_at та review_interval. Така структура дозволяє інтегрувати логіку повторення слів без створення окремої таблиці для flashcards.

Для зберігання результатів quiz-модуля використовується таблиця quiz_attempts. Вона містить інформацію про виконані тестові спроби, правильність відповіді та дату проходження quiz-сценарію. Таблиця пов’язана із users та dictionary_entries через зовнішні ключі, що дозволяє аналізувати результати навчання окремого користувача. Фрагменти реалізації quiz-модуля наведено у додатку Б.

ER-діаграма бази даних демонструє структуру взаємозв'язків між основними сутностями системи та ілюструє реалізацію зв'язків між таблицями (рис. 2.2). Додаткові фрагменти ORM-моделей та зв'язків наведено у додатку Б.

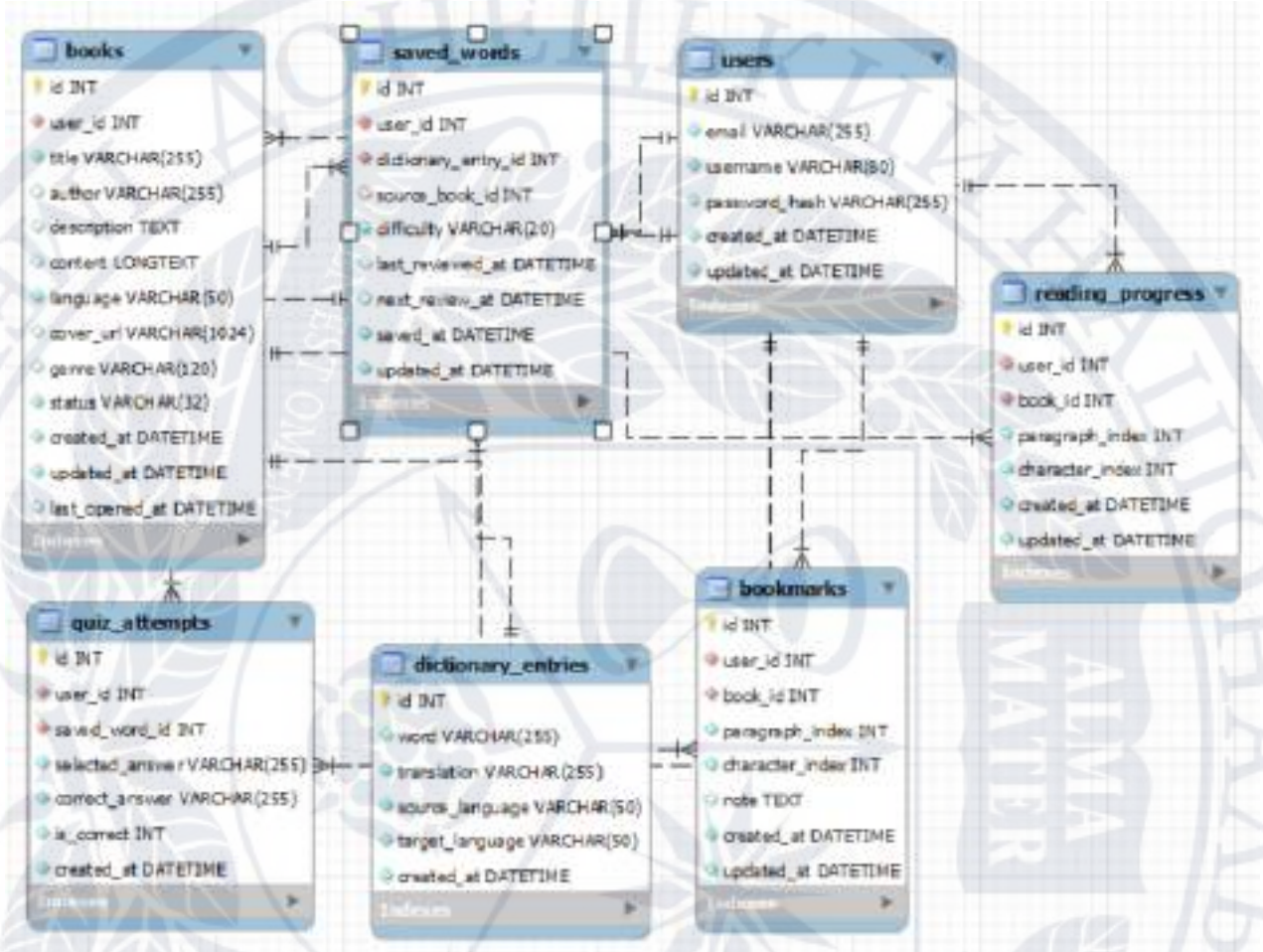

Рисунок 2.2 – ER-діаграма бази даних системи мобільного додатку для читання з інтерактивним словником

Для оптимізації роботи запитів у таблицях використовуються indexes. Індекси створені для полів, які найчастіше беруть участь у WHERE, JOIN та ORDER BY операціях. Зокрема, індексація застосовується для user_id, book_id та dictionary_entry_id. Подібний підхід прискорює пошук пов'язаних записів та зменшує час виконання JOIN-запитів між таблицями [33].

Разом із перевагами індексація має певні компроміси. Кожен додатковий індекс збільшує обсяг пам'яті, необхідної для зберігання БД, а також уповільнює INSERT та UPDATE операції через необхідність оновлення індексних структур.

Саме тому індекси були додані лише для полів, які активно використовуються у фільтрації та побудові зв’язків між сутностями [34].

Для роботи з базою даних у backend-частині використовується SQLAlchemy ORM. ORM-підхід дозволяє представляти записи таблиць у вигляді Python-об’єктів і працювати зі зв’язками між сутностями через зв’язки. У межах проєкту ORM спрощує реалізацію backend-логіки, оскільки більшість операцій із БД виконується через моделі та session API без необхідності написання великої кількості raw SQL-запитів. Приклади реалізації ORM-моделей серверної частини наведено у додатку Б.

Використання SQLAlchemy ORM також позитивно впливає на підтримуваність коду. Опис структури таблиць, зв’язків та constraints зосереджений у моделях, що підтримує узгодженість між backend-кодом і реальною структурою БД. Разом із цим ORM може створювати додаткові накладні витрати для складних або високонавантажених SQL-запитів. У системах із критичними вимогами до продуктивності окремі операції можуть вимагати використання raw SQL, однак для функціоналу бакалаврського проєкту можливостей ORM є достатньо.

Таким чином, спроєктована база даних підтримує структуроване зберігання інформації, забезпечує цілісність даних між сутностями та відповідає функціональним вимогам мобільного додатку. Використання normalization, ForeignKey-залежностей, UNIQUE constraints та indexes створює основу для стабільної роботи backend-системи та подальшого розширення функціоналу застосунку [35].

2.3 Проєктування REST API та механізмів безпеки

Проєктування REST API є одним із центральних етапів розробки серверної частини мобільного додатку, оскільки саме API визначає спосіб взаємодії між Flutter-клієнтом і backend-системою. Архітектура API впливає на масштабованість застосунку, підтримуваність коду, узгодженість передачі даних

та інтеграцію окремих компонентів системи. Для реалізації серверної частини було використано REST-oriented підхід, у межах якого взаємодія між клієнтом і сервером здійснюється через HTTP-запити та JSON-відповіді.

REST API організовано навколо окремих ресурсів системи. Кожна функціональна сутність − користувачі, книги, закладки, словникові записи або quiz-модуль − представлена окремою групою URI-маршрутів. Подібний resource-oriented підхід робить структуру API передбачуваною та спрощує інтеграцію frontend-клієнта із серверною частиною [36].

Для роботи з ресурсами використовуються стандартні HTTP methods. Метод GET застосовується для отримання інформації про ресурси, POST − для створення нових записів, PUT − для оновлення даних, DELETE − для видалення сутностей. Використання стандартної HTTP-семантики підтримує узгодженість API та спрощує роботу клієнтської частини із backend-сервісом [37].

REST-архітектура також базується на принципі stateless communication. Сервер не зберігає інформацію про стан клієнтської сесії між запитами, а кожен HTTP-запит містить усі необхідні дані для його обробки. Stateless-підхід спрощує масштабування backend-системи та зменшує зв’язність між окремими компонентами архітектури. Водночас він підвищує вимоги до механізмів автентифікації, оскільки перевірка доступу виконується для кожного запиту окремо.

Для автентифікації користувачів у системі використовується JWT (JSON Web Token). Після успішного входу користувача сервер генерує Bearer token, який надалі використовується для доступу до захищених endpoint-ів. JWT інтегрується з REST API природним чином, оскільки добре підтримує stateless-архітектуру та не потребує серверного зберігання сесій. Фрагменти реалізації JWT-автентифікації наведено у додатку Б.

JWT-токен містить payload із базовою інформацією про користувача. Зокрема, у токені використовується поле sub, яке зберігає ідентифікатор користувача, а також поле exp, що визначає час завершення дії токена [38]. Під

час обробки захищеного запиту сервер декодує JWT та перевіряє валідність підпису і строк дії токена.

Подібний підхід має декілька переваг для мобільних застосунків. JWT зменшує навантаження на сервер, оскільки backend не зберігає інформацію про активні сесії користувачів. Крім того, Bearer token легко інтегрується з Flutter-клієнтом через HTTP headers [39]. Разом із перевагами JWT має певні обмеження. Stateless-підхід ускладнює реалізацію logout та механізмів відкликання токенів, оскільки токен залишається валідним до завершення строку його дії.

Механізми безпеки системи не обмежуються лише автентифікацією. Важливою складовою backend-логіки є контроль авторизації − перевірка прав доступу до конкретних ресурсів. У системі реалізовано ownership validation, відповідно до якої користувач може працювати лише з власними книгами, закладками та словниковими записами.

Для централізованої перевірки авторизації використовується dependency get_current_user. Під час виконання захищеного endpoint-а сервер автоматично перевіряє Bearer token, декодує JWT та отримує інформацію про поточного користувача. Подібний підхід спрощує реалізацію authorization logic та підтримує узгодженість між окремими модулями backend-системи. Приклад реалізації dependency get_current_user наведено у додатку Б.

Додатково для роботи з ресурсами використовується механізм ownership validation через helper-функції на кшталт get_owned_book. Перед виконанням операції сервер перевіряє, чи належить запитувана книга поточному користувачеві. Якщо запис не знайдено або належить іншому користувачу, сервер повертає відповідну помилку доступу. Подібний підхід дозволяє ізолювати приватні дані користувачів та запобігає несанкціонованому доступу до ресурсів системи [40]. Приклад реалізації ownership validation для перевірки доступу до книги наведено у додатку Б.

Окрему увагу під час проєктування механізмів безпеки було приділено зберіганню паролів. Паролі користувачів не записуються у базу даних у відкритому вигляді. Для цього використовується bcrypt hashing. Перед

збереженням password проходить процедуру хешування, після чого в БД записується лише hash-значення [41].

Використання bcrypt підвищує безпеку системи навіть у випадку компрометації бази даних, оскільки відновлення початкового пароля з hash-значення є обчислювально складним завданням. Крім того, bcrypt автоматично використовує salt, що ускладнює застосування rainbow table attacks [40].

Для передачі та валідації даних у backend-системі використовуються Pydantic DTO (Data Transfer Objects). Pydantic-моделі застосовуються для опису структури HTTP-запитів і відповідей, а також автоматичної перевірки типів даних. Фрагменти Pydantic DTO-схем наведено у додатку Б.

Request validation дозволяє перевіряти коректність даних ще до виконання бізнес-логіки endpoint-а. Наприклад, сервер автоматично перевіряє наявність обов'язкових полів, відповідність типів даних та структуру JSON-запиту. У випадку некоректного формату FastAPI повертає validation error без необхідності ручної реалізації перевірок.

Response validation також відіграє важливу роль у підтримці узгодженості API. Pydantic DTO гарантують, що клієнт отримує дані у визначеному форматі незалежно від внутрішньої структури ORM-моделей або database session. Подібне розділення між ORM та DTO позитивно впливає на підтримуваність backend-коду [42].

Для інтеграції функціоналу перекладу використовується зовнішній Translation API. Backend-сервер виступає проміжним рівнем між Flutter-клієнтом і стороннім сервісом перекладу. Запит на переклад надходить до FastAPI, після чого сервер звертається до зовнішнього API та повертає клієнту уніфіковану JSON-відповідь. Приклади інтеграції із зовнішнім Translation API наведено у додатку Б.

Для виконання HTTP-запитів використовується бібліотека httpx. Вона підтримує асинхронну взаємодію із зовнішніми сервісами та добре інтегрується з FastAPI. Під час надсилання запитів до Translation API використовується

timeout configuration, що запобігає надмірному блокуванню backend-потоку у випадку повільної відповіді стороннього сервісу [43].

Параметри інтеграції з Translation API винесені у змінні середовища та зберігаються у файлі .env. До них належать URL зовнішнього сервісу, timeout-параметри та інші конфігураційні значення. Подібний підхід спрощує зміну конфігурації без модифікації вихідного коду backend-системи та зменшує ризик випадкового розміщення конфіденційних даних у репозиторії проєкту.

Під час проєктування REST API та механізмів безпеки основний акцент зроблено на підтримці modular architecture, узгодженості між компонентами системи та ізоляції відповідальностей між рівнями backend-логіки. Використання JWT, ownership validation, Pydantic validation та dependency injection формує достатній рівень безпеки для реалізації функціоналу мобільного додатку та створює основу для подальшого масштабування backend-системи [44].

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ДОДАТКУ

3.1 Реалізація серверної архітектури

Серверна частина мобільного додатку для читання з вбудованим інтерактивним словником реалізована у вигляді REST API на основі фреймворку FastAPI. Використання REST-архітектури є доцільним для мобільного застосунку, оскільки Flutter-клієнт взаємодіє із сервером через стандартизовані HTTP-запити та отримує відповіді у форматі JSON. Поділ frontend і backend на незалежні компоненти спрощує підтримку системи та дає змогу модифікувати клієнтську або серверну частину без необхідності суттєвих змін у суміжних модулях. Flutter-клієнт відповідає за взаємодію з користувачем та відображення інтерфейсу, тоді як серверна частина виконує автентифікацію, обробку даних, взаємодію з базою даних, роботу зі словником і реалізацію навчального функціоналу.

Архітектурно система складається з чотирьох основних компонентів: Flutter-клієнта, FastAPI-сервера, реляційної бази даних MySQL та зовнішнього сервісу перекладу. Мобільний застосунок надсилає HTTP-запити до REST API, сервер обробляє їх відповідно до логіки окремих модулів, взаємодіє з MySQL через SQLAlchemy ORM та, за потреби, виконує запити до зовнішнього Translation API для отримання перекладу слів. Подібний поділ відповідальностей спрощує масштабування системи та ізолює окремі компоненти один від одного.

На рисунку 3.1 подано загальну архітектуру взаємодії між Flutter-клієнтом, FastAPI-сервером, базою даних MySQL та зовнішнім Translation API.

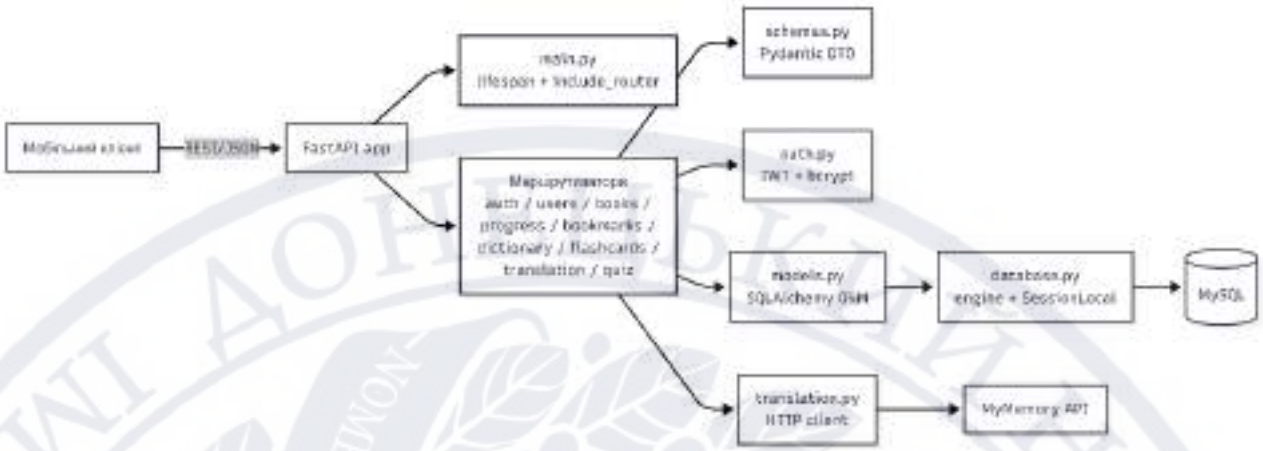

Рисунок 3.1 – Загальна архітектура системи

REST-підхід дозволяє представляти основні об’єкти системи у вигляді окремих ресурсів: користувачів, книг, прогресу читання, закладок, словникових записів, збережених слів, flashcards та quiz-запитань. Для роботи з ресурсами використовуються стандартні HTTP-методи. Наприклад, створення книги виконується через POST-запит, отримання списку книг − через GET, оновлення − через PUT, а видалення − через DELETE. Така структура робить API передбачуваним для клієнтської частини та спрощує інтеграцію з мобільним застосунком.

Точкою входу серверного застосунку є файл main.py, у якому створюється екземпляр FastAPI, задаються базові параметри API та підключаються функціональні router-модулі. У цьому ж файлі реалізовано службовий endpoint /health, призначений для перевірки працездатності серверної частини. Під час запуску застосунку виконується ініціалізація таблиць бази даних через Base.metadata.create_all(bind=engine), що спрощує розгортання системи у середовищі розробки.

Структура backend-проєкту організована за модульним принципом. Окремими router-модулями винесено авторизацію, роботу з користувачами, книги, прогрес читання, закладки, словник, переклад, flashcards та quiz. Фрагменти коду ініціалізації FastAPI-застосунку та підключення router-модулів наведено у додатку Б. Подібна організація коду зменшує зв’язність між компонентами системи та спрощує підтримку backend-частини. Наприклад,

зміни в алгоритмі повторення слів для flashcards не впливають на модуль авторизації або CRUD-операції для книг. Модульна архітектура також спрощує масштабування системи та подальше розширення функціоналу. На рисунку 3.2 зображено структуру backend-проєкту у середовищі Visual Studio Code.

Рисунок 3.2 – Структура backend-проєкту у середовищі Visual Studio Code.

Підключення до бази даних винесено в окремий файл database.py. Для роботи з MySQL використовується SQLAlchemy, у межах якого створюється engine, налаштовується SessionLocal та визначається базовий клас Base для ORM-моделей. Функція get_db() використовується як dependency FastAPI: для кожного HTTP-запиту створюється окрема сесія взаємодії з базою даних, яка автоматично закривається після завершення обробки запиту [45]. Використання dependency injection у цьому випадку зменшує дублювання коду між endpoint-ами та спрощує контроль життєвого циклу database session.

SQLAlchemy ORM використовується для представлення структури бази даних у вигляді Python-моделей. У серверній частині реалізовано моделі User, Book, ReadingProgress, Bookmark, DictionaryEntry, SavedWord та QuizAttempt. Між сутностями налаштовано ForeignKey-зв'язки та relationship-відношення, що дозволяє працювати з пов'язаними даними на рівні об'єктної моделі. Наприклад, користувач має пов'язані книги, закладки, прогрес читання, збережені слова та результати проходження quiz. Використання ORM-підходу зменшує кількість ручного SQL-коду та робить структуру серверної логіки більш зрозумілою.

Порівняно з використанням raw SQL, SQLAlchemy ORM спрощує підтримку проєкту та зменшує ризик помилок під час роботи зі зв'язками і транзакціями. Водночас ORM-підхід може бути менш ефективним для складних аналітичних запитів або високонавантажених систем, де необхідний повний контроль над SQL-запитами [46].

Для опису структури HTTP-запитів і відповідей використовуються Pydantic-схеми, які виконують роль DTO-об'єктів між клієнтом і сервером. Вони відповідають за валідацію типів даних, формування JSON-відповідей та контроль структури API. Наприклад, для книг визначено окремі схеми створення, оновлення та читання, а для словника − схеми збереження слова та повернення словникового запису. Подібне розмежування важливе з точки зору архітектури, оскільки ORM-моделі описують структуру таблиць бази даних, а Pydantic-схеми − формат взаємодії через REST API [47].

Безпековий рівень системи побудовано на JWT-автентифікації. Після успішного входу користувач отримує access token, який надалі передається у заголовку Authorization у форматі Bearer. Серверна частина не зберігає стан сесії між запитами, а виконує перевірку JWT під час кожного звернення до захищеного endpoint-а. Stateless-підхід спрощує масштабування REST API та усуває необхідність зберігання серверних сесій.

У серверному коді використовується HTTPBearer, а перевірка токена виконується у функції get_current_user. JWT містить поле sub з ідентифікатором користувача та поле exp, яке визначає строк дії токена. Централізована перевірка

автентифікації через dependency injection спрощує повторне використання логіки авторизації між endpoint-ами та зменшує дублювання коду [48].

Разом із перевагами JWT-підхід має певні обмеження. Stateless-архітектура ускладнює реалізацію механізмів logout та дострокового відкликання токенів, оскільки сервер не зберігає інформацію про активні сесії. У production-системах для розв'язання цієї проблеми зазвичай використовуються refresh token або blacklist-механізми.

Паролі користувачів не зберігаються у відкритому вигляді. Для їхнього захисту використовується bcrypt-хешування через бібліотеку passlib. Під час реєстрації пароль перетворюється на хеш, а під час входу введене значення порівнюється зі збереженим хешем. Навіть у разі несанкціонованого доступу до бази даних це ускладнює отримання реальних паролів користувачів. Фрагменти реалізації JWT-автентифікації, хешування паролів та отримання поточного користувача наведено у додатку Б.

Конфігураційні параметри серверної частини винесені в змінні середовища. До них належать рядок підключення до бази даних, секретний ключ JWT, алгоритм підпису токена, строк його дії, URL зовнішнього Translation API та timeout для HTTP-запитів. Використання .env-файлу дозволяє не зберігати конфіденційні параметри безпосередньо у вихідному коді та спрощує перенесення системи між середовищами розробки, тестування та розгортання [49].

На рисунку 3.3 представлено Component Diagram серверної частини системи, яка відображає взаємодію між Flutter-клієнтом, FastAPI-сервером, модулем автентифікації, ORM-рівнем, базою даних та Translation API.

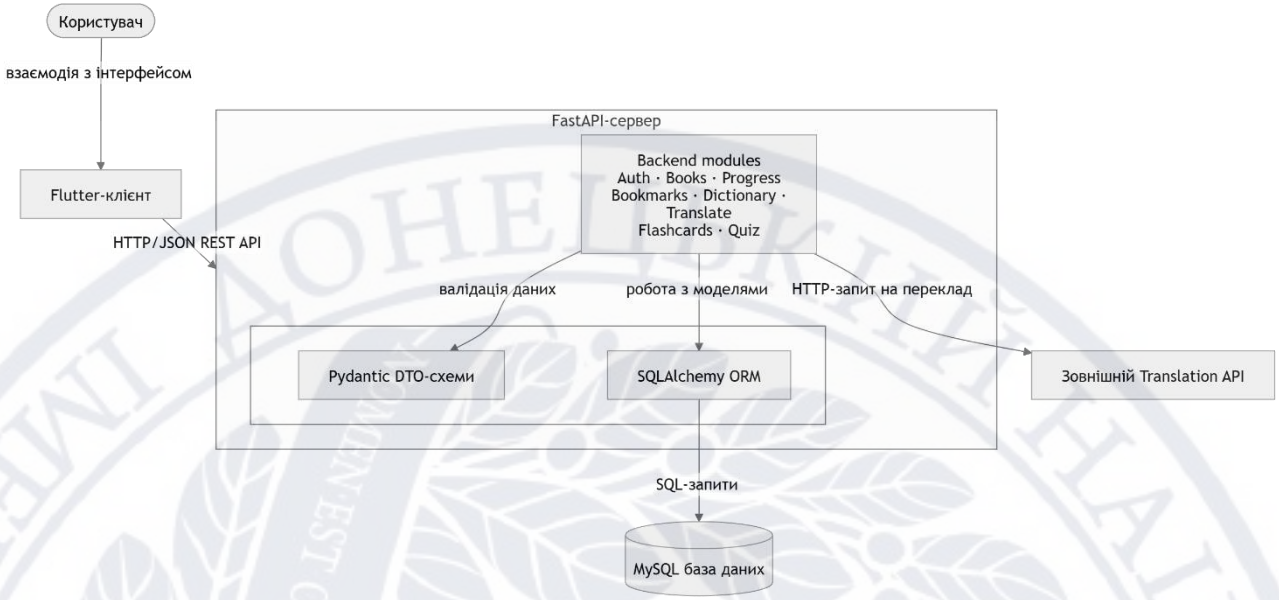

Рисунок 3.3 – Component Diagram серверної архітектури

3.2 Реалізація функціональних модулів системи

Функціональні модулі серверної частини реалізовані як окремі router-и FastAPI. Кожен модуль відповідає за визначену групу сценаріїв: авторизацію, роботу з користувачем, керування книгами, збереження прогресу, створення закладок, переклад слів, ведення персонального словника, повторення лексики та проходження quiz. Такий розподіл відповідає принципу розділення відповідальностей і полегшує подальше розширення системи. Повний перелік endpoint-ів REST API серверної частини подано у додатку А.

Структура бази даних побудована відповідно до третьої нормальної форми. Основні сутності системи − користувачі, книги, прогрес читання, закладки, словникові записи та результати quiz − винесені в окремі таблиці, пов’язані між собою через зовнішні ключі. Подібний підхід зменшує дублювання даних і спрощує підтримку цілісності бази даних. Для оптимізації пошуку та JOIN-операцій використовуються індекси для полів user_id, book_id, dictionary_entry_id та інших полів, які часто застосовуються у WHERE-запитах.

Зв’язки між таблицями реалізовані через SQLAlchemy relationships. Наприклад, користувач пов’язаний із книгами, прогресом читання, закладками

та збереженими словами. Для окремих залежностей використовується ON DELETE CASCADE, що дозволяє автоматично видаляти пов’язані записи після видалення користувача або книги. Каскадне видалення спрощує підтримку цілісності даних бази даних, однак потребує обережного проєктування зв’язків, оскільки видалення parent-сутності автоматично впливає на дочірні записи. Детальну структуру взаємозв’язків між таблицями та ER-модель бази даних наведено в підрозділі 2.2.

Модуль авторизації відповідає за створення облікового запису, перевірку облікових даних та формування JWT-токена. Endpoint POST /auth/register приймає email, username та password, після чого сервер перевіряє унікальність користувача й створює новий запис у таблиці users:

```
POST /auth/register
{
"email": "user@example.com",
"username": "reader1",
"password": "password123"
}
```

Під час реєстрації пароль не зберігається у відкритому вигляді. Перед записом до бази даних він хешується через bcrypt. Навіть у разі несанкціонованого доступу до БД це ускладнює отримання реальних облікових даних користувачів

Після успішної авторизації endpoint POST /auth/login повертає JWT-токен доступу:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer"
}
```

На рисунку 3.5 представлено приклад отримання Bearer token після успішного виконання запиту POST /auth/login у Swagger UI.

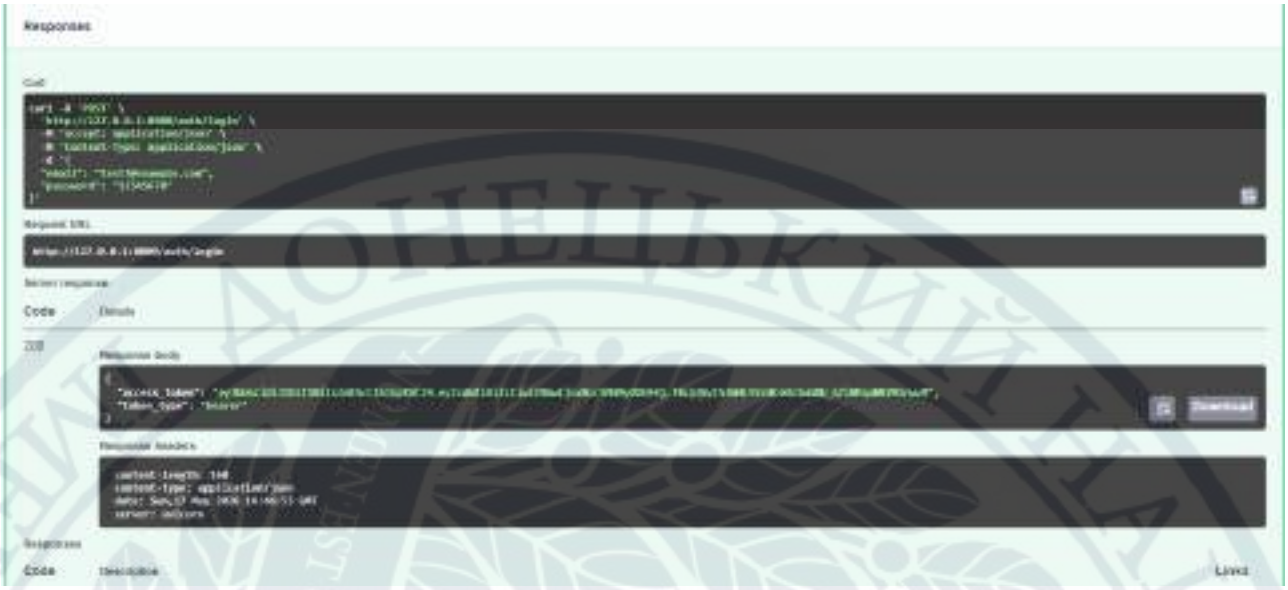

Рисунок 3.5 – Приклад отримання Bearer token після успішного виконання запиту POST /auth/login у Swagger UI

Отриманий токен використовується для всіх захищених endpoint-ів. На рівні backend перевірка авторизації виконується через dependency injection та функцію get_current_user. Сервер отримує Bearer token із заголовка Authorization, декодує JWT, зчитує ідентифікатор користувача з поля sub та завантажує відповідний запис із таблиці users. Якщо токен відсутній, пошкоджений або прострочений, FastAPI повертає статус-код 401 Unauthorized.

JWT-підхід добре підходить для REST API мобільного застосунку, оскільки не потребує зберігання серверних сесій і спрощує stateless-взаємодію між Flutter-клієнтом і backend-частиною. Водночас подібна архітектура ускладнює реалізацію logout та механізмів відкликання токенів, оскільки сервер не зберігає централізований список активних сесій.

На рисунку 3.6 зображено Sequence Diagram процесу JWT-автентифікації.

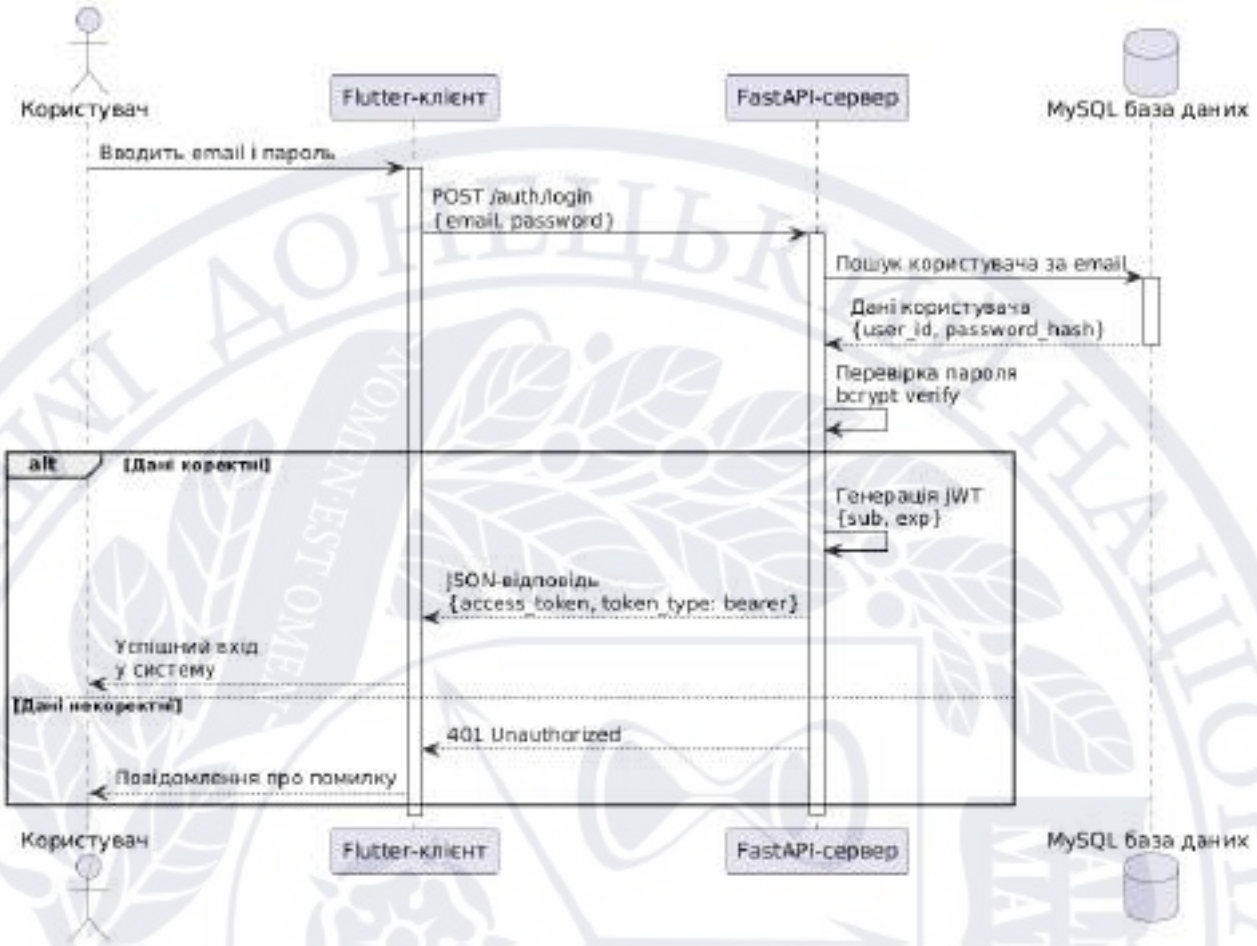

Рисунок 3.6 – Sequence Diagram процесу JWT-автентифікації

Модуль користувача представлений endpoint-ом GET /users/me. Його призначення полягає в отриманні інформації про поточного авторизованого користувача. Flutter-клієнт використовує цей endpoint для перевірки дійсності токена після входу в систему та отримання базових даних профілю.

Модуль книг реалізує CRUD-операції над персональною бібліотекою користувача. ORM-модель Book містить назву книги, автора, опис, текстовий вміст, мову, жанр, обкладинку, статус та службові часові поля. Кожна книга пов’язана з конкретним користувачем через поле user_id, тому бібліотеки різних користувачів ізольовані між собою.

Створення книги виконується через endpoint POST /books:

```
POST /books
{
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
```

```
  "description": "A novel set in the 1920s",
  "content": "...",
  "language": "en"
}
```

Після створення книги сервер повертає сформований запис із власним ідентифікатором та службовими часовими полями. Подібний endpoint демонструє взаємодію між Pydantic-схемами, ORM-моделлю Book та таблицею books.

На рисунку 3.7 подано приклад створення книги через Swagger UI.

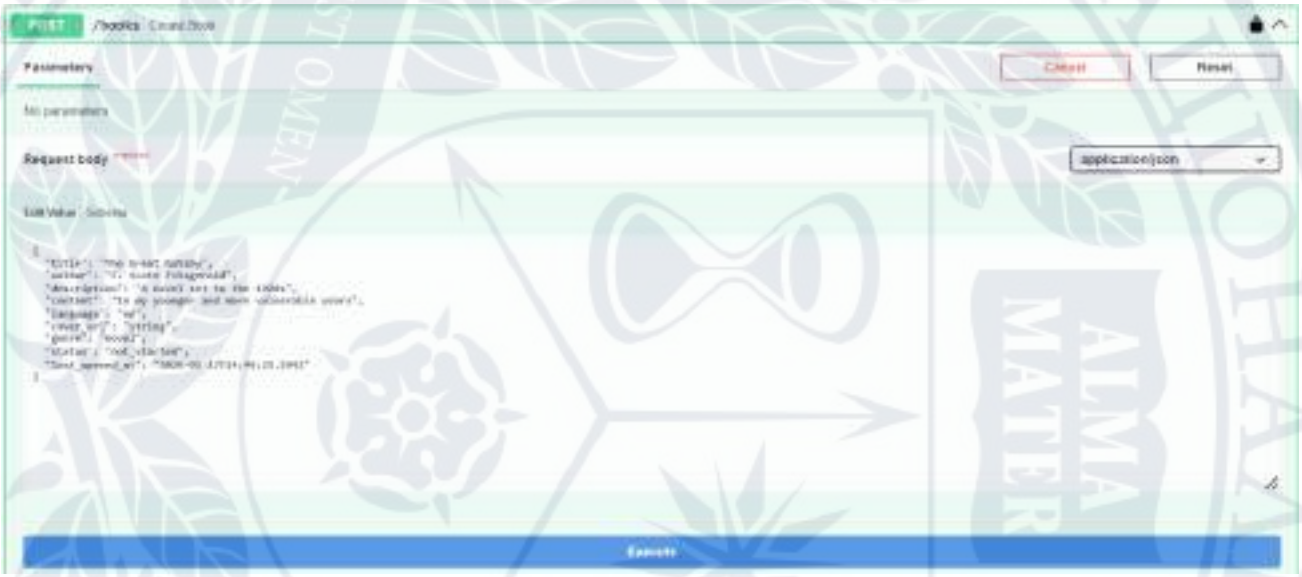

Рисунок 3.7 – Приклад створення книги

Для запобігання доступу до чужих ресурсів у модулі книг використовується функція get_owned_book. Вона перевіряє не лише існування книги, а й належність цієї книги поточному користувачу. Такий механізм реалізує контроль авторизації на рівні backend і захищає приватну бібліотеку користувача. Приклад реалізації ownership validation наведено у додатку Б.

Модуль прогресу читання взаємодіє з таблицею reading_progress, де для кожного користувача та кожної книги зберігається поточна позиція у вигляді індексів абзацу та символу. Такий підхід забезпечує незалежність фіксації прогресу від розміру екрана чи параметрів відображення мобільного клієнта. Оновлення даних на сервері здійснюється шляхом надсилання HTTP-запиту за наступним форматом:

```
PUT /books/1/progress
{
  "paragraph_index": 25,
  "character_index": 120
}
```

У таблиці reading_progress використовується UNIQUE constraint для пари user_id і book_id. Завдяки цьому для однієї книги конкретного користувача існує лише один запис прогресу. Подібне обмеження спрощує оновлення даних та запобігає дублюванню записів.

На рисунку 3.8 представлено приклад оновлення прогресу читання.

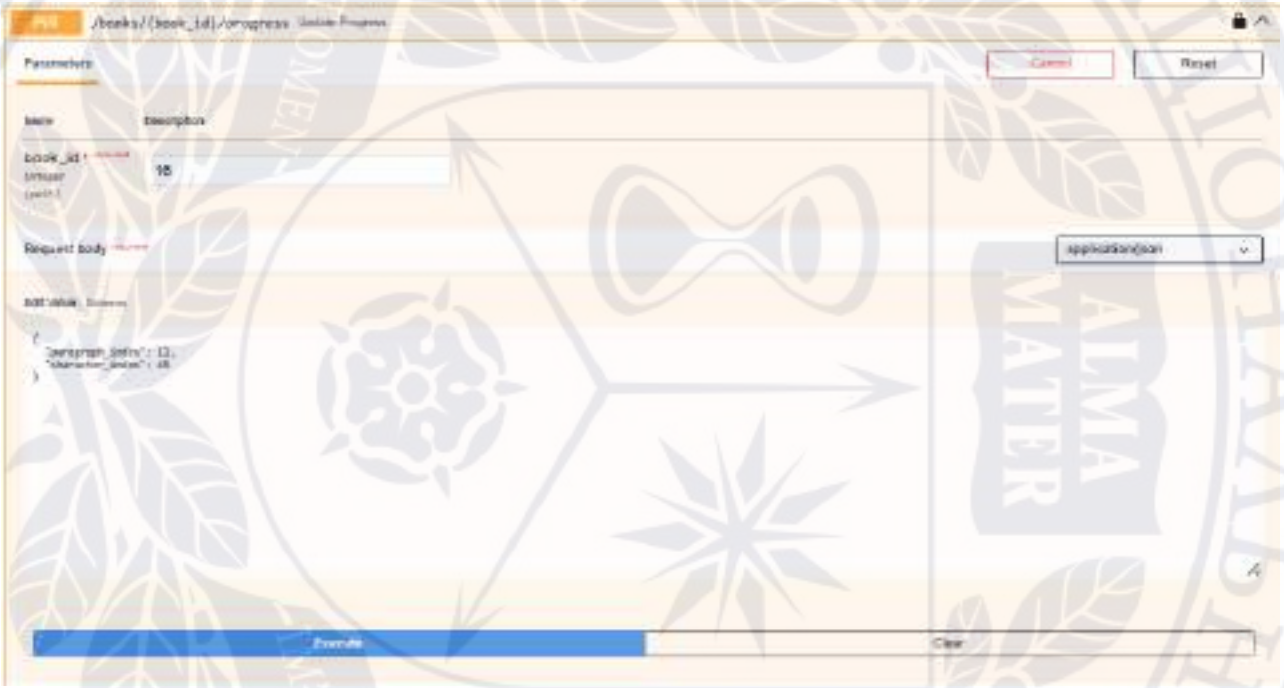

Рисунок 3.8 – Приклад оновлення прогресу читання

Модуль закладок дозволяє користувачу зберігати важливі позиції у тексті книги. Закладка пов’язана з користувачем та книгою через ForeignKey-залежності й містить позицію у тексті та необов’язкову нотатку.

```
POST /books/1/bookmarks
{
  "paragraph_index": 12,
  "character_index": 45,
  "note": "Important quote"
}
```

Перед створенням або отриманням закладок сервер перевіряє доступ до книги через get_owned_book. Користувач не може переглядати або змінювати закладки для чужої книги.

На рисунку 3.9 подано приклад створення закладки.

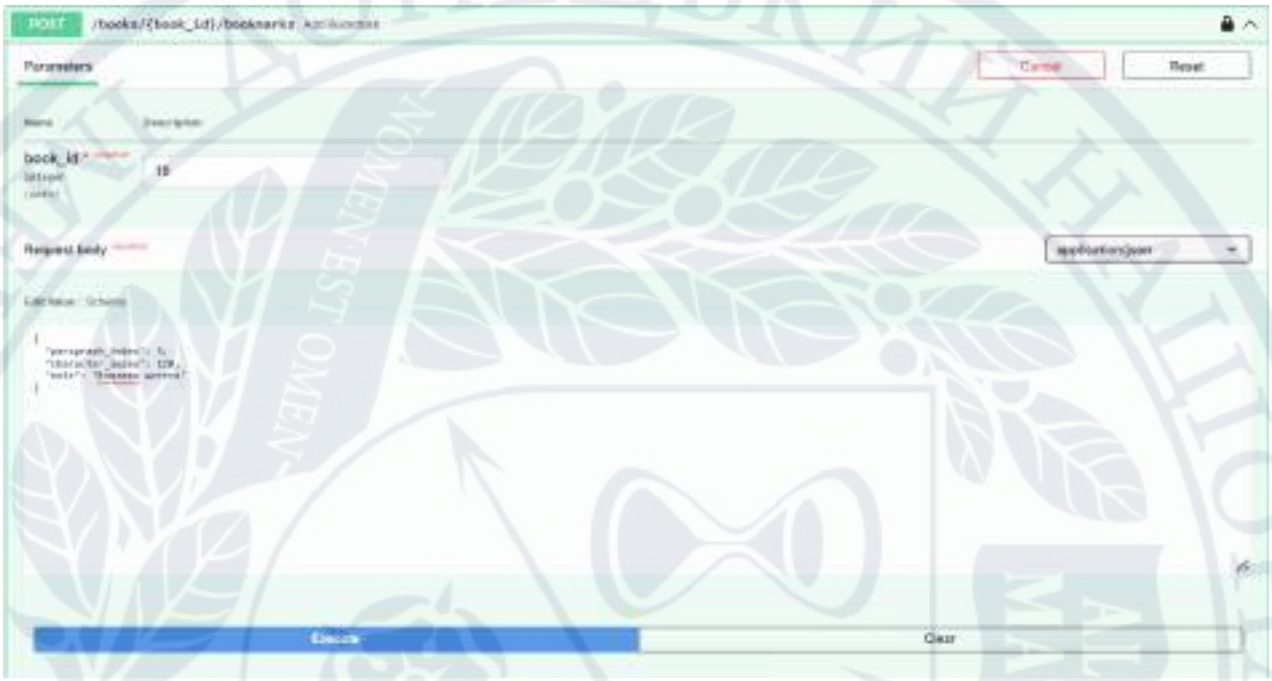

Рисунок 3.9 – Приклад створення закладки

Модуль перекладу відповідає за взаємодію із зовнішнім Translation API.

Endpoint POST /translate приймає слово, мову оригіналу та мову перекладу:

```
POST /translate
{
  "word": "book",
  "source_language": "en",
  "target_language": "uk"
}
```

Для виконання HTTP-запитів використовується бібліотека httpx. URL зовнішнього сервісу та timeout для запитів задаються через змінні середовища. Подібне рішення спрощує зміну Translation API без модифікації основної логіки backend.

Інтеграція із зовнішнім сервісом перекладу дозволила реалізувати функціонал live-перекладу без створення власної системи машинного перекладу.

Водночас backend стає залежним від доступності стороннього API та стабільності мережевого з’єднання.

На рисунку 3.10 зображено Sequence Diagram сценарію отримання перекладу слова.

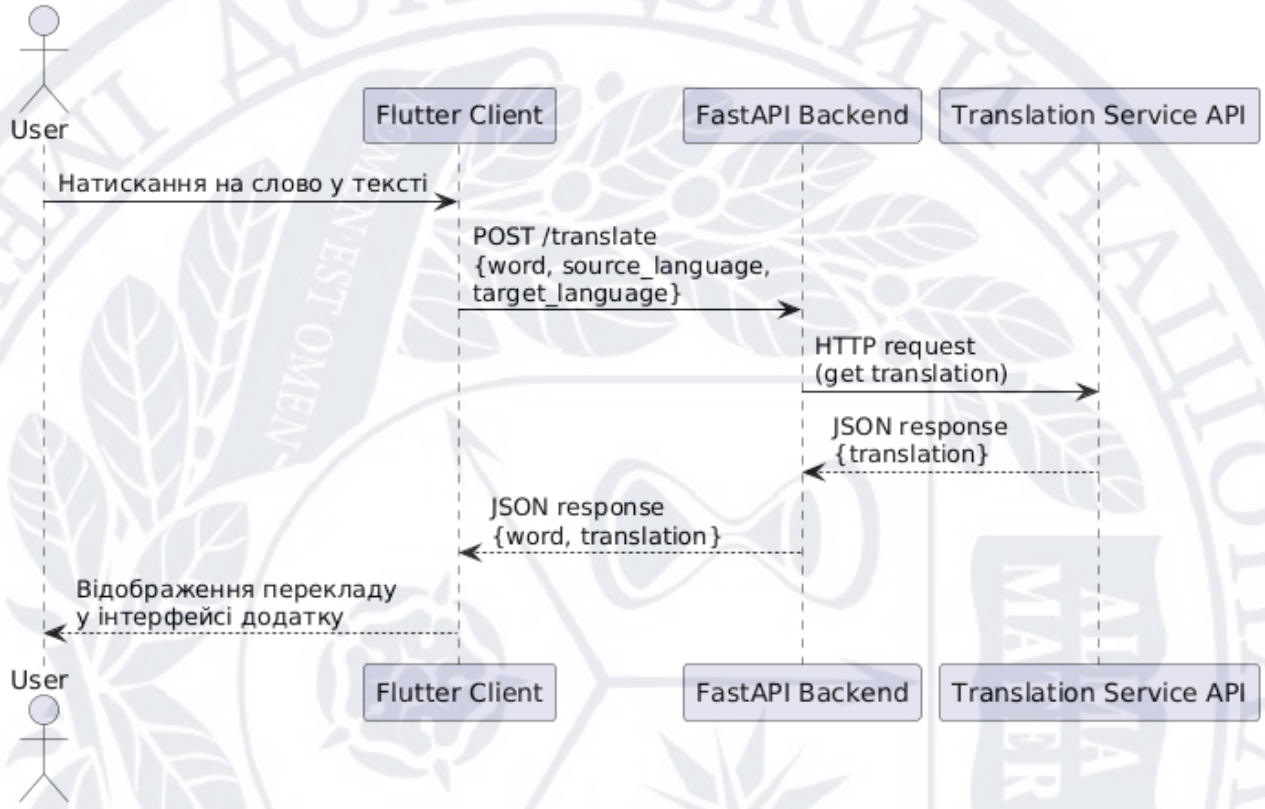

Рисунок 3.10 – Sequence Diagram сценарію отримання перекладу слова

Модуль словника реалізований через таблиці dictionary_entries та saved_words. Таблиця dictionary_entries містить унікальні словникові записи, тоді як saved_words відповідає за зв’язок між користувачем та словом.

Для уникнення дублювання використовується UNIQUE constraint для комбінації word, source_language та target_language. Аналогічно у таблиці saved_words використовується UNIQUE constraint для пари user_id і dictionary_entry_id, тому користувач не може зберегти одне й те саме слово кілька разів.

Під час отримання персонального словника застосовується joinedload, що дозволяє оптимізувати завантаження пов’язаних словникових записів та зменшити кількість SQL-запитів до БД.

Модуль flashcards використовується для повторення збереженої лексики. Логіка повторення базується на спрощеному підході інтервального повторення. Для кожного слова зберігаються current_interval, next_review_at та difficulty. Після успішного повторення інтервал збільшується, а дата наступного повторення переноситься вперед [50]. Фрагменти реалізації flashcards-модуля наведено у додатку Б.

```
PUT /flashcards/1
{
  "difficulty": "easy"
}
```

Подібний механізм дозволяє адаптувати інтервал повторення залежно від складності слова та результатів попередніх відповідей користувача [51].

Модуль quiz використовується для перевірки знань користувача. Endpoint GET /quiz/question генерує питання та декілька варіантів відповідей. Для формування quiz використовуються слова із персонального словника користувача.

```
GET /quiz/question
{
  "question_id": 15,
  "word": "book",
  "options": ["книга", "ручка", "стіл", "вікно"]
}
```

Після вибору відповіді Flutter-клієнт надсилає POST /quiz/answer:

```
POST /quiz/answer
{
  "question_id": 15,
  "selected_answer": "книга"
}
```

Сервер перевіряє правильність відповіді та зберігає результат у таблиці quiz_attempts. Подібний підхід дозволяє вести історію проходження quiz та використовувати результати для подальшої адаптації flashcards-механізму.

На рівні ORM-моделей між saved_words та quiz_attempts налаштовані relationship-залежності. Для полів user_id та saved_word_id використовуються indexes, що пришвидшує пошук статистики та історії проходження quiz.

Таким чином, реалізовані функціональні модулі охоплюють повний цикл роботи користувача із системою: від авторизації та читання книг до перекладу слів, ведення персонального словника, повторення лексики та проходження навчальних quiz-сценаріїв.

3.3 Мануальна верифікація API та інтеграції серверної і клієнтської частин

У межах даного підрозділу проведено мануальну верифікацію серверної та клієнтської частин системи. Робота була спрямована на перевірку коректності функціонування REST API, механізмів авторизації, взаємодії з базою даних, обробки помилок та інтеграції з Flutter-клієнтом. Основна увага приділялася функціональній перевірці endpoint-ів, контролю JWT-автентифікації, CRUD-операцій та стабільності взаємодії між клієнтською і серверною частинами системи. Використання мануального підходу дозволило провести детальний аналіз динамічної взаємодії між компонентами системи на різних етапах обробки запитів.

Для верифікації API використовувався Swagger UI, який автоматично генерується FastAPI на основі опису endpoint-ів і Pydantic-схем. Використання Swagger UI спрощує перевірку HTTP-запитів, оскільки дозволяє надсилати запити безпосередньо з браузера, переглядати JSON-відповіді та аналізувати HTTP status codes. Крім цього, Swagger UI автоматично відображає структуру request body, параметри endpoint-ів та схеми відповідей, що спрощує перевірку узгодженості між backend-кодом і REST API [52].

На рисунку 3.11 подано Swagger UI серверної частини системи

Рисунок 3.11 – Swagger UI серверної частини системи

Процес верифікації розпочинався з перевірки endpoint-ів авторизації. Під час перевірки POST /auth/register очікуваним результатом було створення нового користувача у таблиці users та повернення статус-коду 201 Created. Фактичний результат підтвердив коректне створення запису користувача та хешування пароля перед збереженням у базі даних.

Під час верифікації POST /auth/login перевірялася правильність формування JWT-токена. Очікуваним результатом було повернення access_token та token_type у форматі bearer. Сервер успішно формував JWT після перевірки email і password користувача.

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR...",
  "token_type": "bearer"
}
```

Отриманий Bearer token використовувався для доступу до захищених endpoint-ів через кнопку Authorize у Swagger UI. На рисунку 3.12 представлено приклад авторизації через Bearer token.

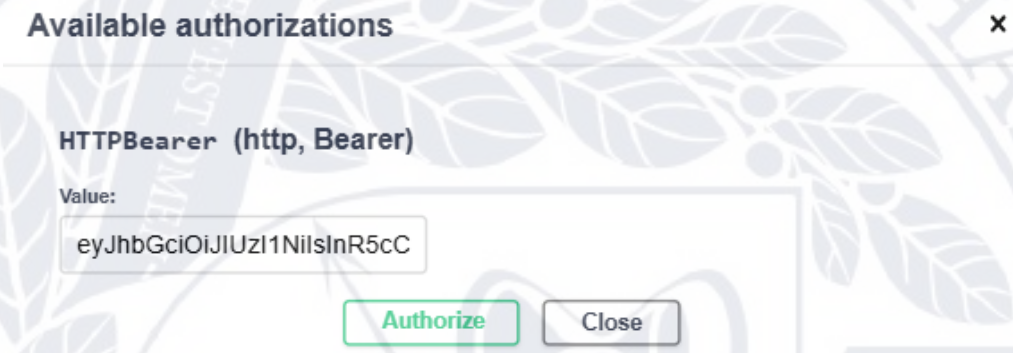

Рисунок 3.12 – Авторизація через Bearer token у Swagger UI

Під час веифікації JWT-автентифікації окремо перевірялися негативні сценарії. Якщо Authorization header був відсутній або містив некоректний токен, сервер повертав статус-код 401 Unauthorized. Аналогічний результат спостерігався для прострочених або пошкоджених JWT. Подібна поведінка підтверджує коректність роботи dependency injection та функції get_current_user, яка централізовано виконує перевірку авторизації для захищених endpoint-ів.

Після перевірки авторизації тестувалися CRUD-операції модуля книг. Для endpoint-а POST /books очікуваним результатом було створення нового запису у таблиці books та повернення статус-коду 201 Created. Під час локального тестування сервер коректно створював ORM-об’єкт Book, записував його у базу даних та повертав JSON-відповідь із generated id, user_id та часовими полями.

Функціональна верифікація підтвердила коректну роботу endpoint-ів GET /books, PUT /books/{book_id} та DELETE /books/{book_id}. Під час оновлення книги перевірялася синхронізація між Pydantic-схемами та ORM-моделлю Book. Для неіснуючих ідентифікаторів сервер повертав статус-код 404 Not Found.

Окремо тестувався механізм ownership validation. Під час звернення до книги іншого користувача функція get_owned_book блокувала доступ до ресурсу та повертала помилку 404 Not Found. Подібний підхід приховує інформацію про існування чужих ресурсів і підсилює контроль авторизації на рівні backend.

Верифікація endpoint-ів прогресу читання та закладок підтвердило правильність роботи зв між таблицями books, reading_progress та bookmarks. Для endpoint-а PUT /books/{book_id}/progress перевірялося оновлення paragraph_index і character_index для конкретного користувача та книги. Очікуваним результатом було оновлення існуючого запису без створення дублікатів, що підтверджувало коректність UNIQUE constraint для пари user_id і book_id.

Під час верифікації закладок перевірялися створення, отримання та видалення bookmark-записів. Сервер коректно виконував контроль авторизації та не дозволяв створювати закладки для книг інших користувачів.

Для endpoint-а POST /translate тестувалася інтеграція із зовнішнім Translation API. Очікуваним результатом було отримання перекладу слова та повернення JSON-відповіді Flutter-клієнту. Під час тестування сервер коректно виконував HTTP-запит через бібліотеку httpx та обробляв відповідь стороннього сервісу.

Окремо перевірялися сценарії помилок. Якщо зовнішній Translation API був недоступний або перевищував timeout, backend коректно обробляв винятки та повертав помилку без аварійного завершення застосунку. Використання timeout для HTTP-запитів зменшує ризик зависання серверного endpoint-а під час очікування відповіді від стороннього сервісу.

Верифікація модулів flashcards та quiz виконувалося окремо. Для GET /flashcards перевірялося отримання слів, дата повторення яких є меншою або рівною поточному часу. Під час оновлення flashcard через PUT /flashcards/{id} перевірялася зміна current_interval, next_review_at та difficulty.

Для quiz-модуля перевірялися генерація питання, формування варіантів відповідей та збереження результатів у таблиці quiz_attempts. Endpoint GET

/quiz/question коректно повертав question_id, слово та набір відповідей. Після надсилання POST /quiz/answer сервер перевіряв правильність вибору користувача та зберігав результат проходження тесту.

Окремо перевірялися сценарії некоректних запитів. Якщо request body не відповідав Pydantic-схемі або містив неправильний тип даних, FastAPI автоматично повертав статус-код 422 Validation Error. Подібна поведінка підтверджує коректність використання Pydantic DTO для валідації вхідних даних.

Після завершення функціональної верифікації API виконувалося інтеграційне тестування Flutter-клієнта та FastAPI-сервера. Для цього використовувався ngrok, який створював тимчасовий публічний HTTPS-тунель до локального FastAPI-застосунку. Flutter-клієнт взаємодіяв із backend через URL ngrok, що дозволяло тестувати мобільний застосунок на фізичному пристрої без розгортання сервера у production-середовищі [53].

На рисунку 3.13 представлено Sequence Diagram взаємодії Flutter-клієнта з FastAPI-сервером через ngrok.

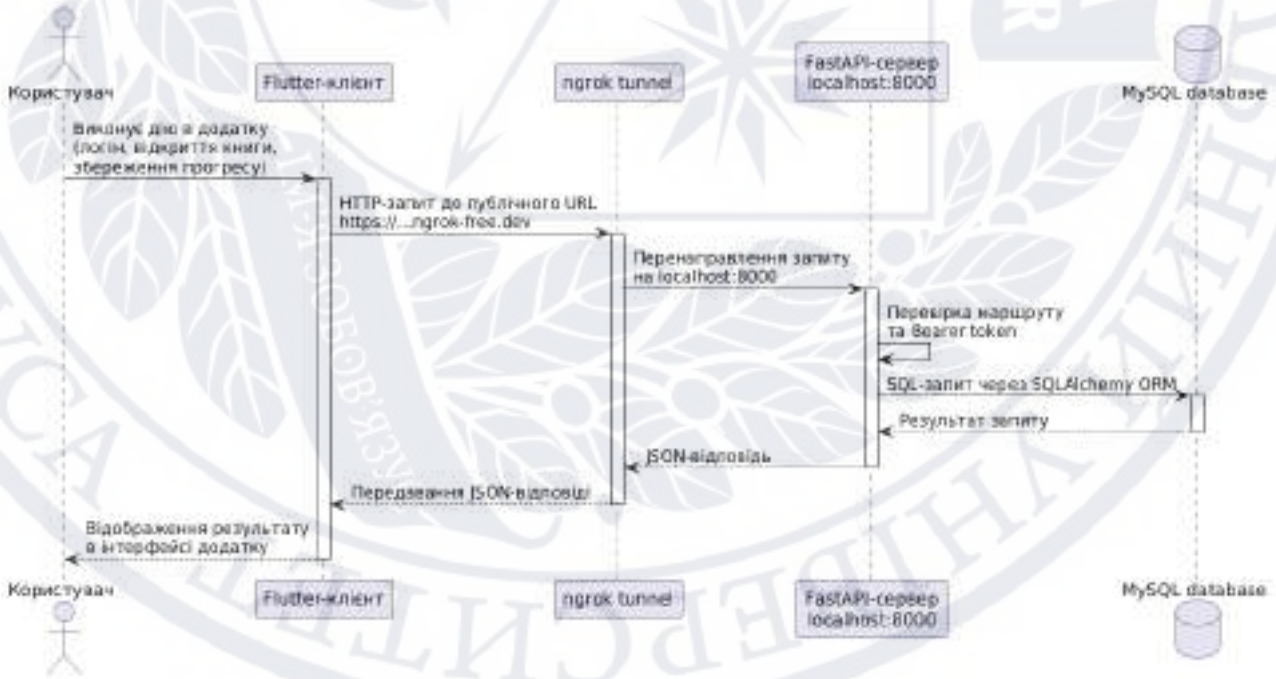

Рисунок 3.13 – Sequence Diagram взаємодії Flutter-клієнта з FastAPI-сервером через ngrok

Під час інтеграційного тестування перевірялися:

- авторизація користувача;
- отримання списку книг;
- створення книг;
- оновлення прогресу читання;
- створення закладок;
- робота перекладу;
- взаємодія зі словником;
- робота flashcards;
- проходження quiz.

Фактичні результати підтвердили коректну взаємодію Flutter-клієнта з FastAPI-сервером через REST API. Flutter успішно отримував JSON-відповіді, обробляв HTTP status codes та синхронізував стан інтерфейсу з даними backend.

Водночас під час інтеграційного тестування були виявлені обмеження використання ngrok. Оскільки безкоштовна версія сервісу створює тимчасові URL, після перезапуску тунелю Flutter-клієнт потребував оновлення адреси backend. Крім цього, ngrok є інструментом локальної розробки й не призначений для production-використання, тому для реального розгортання системи необхідне використання окремого сервера або хмарної інфраструктури.

Таким чином, проведене тестування підтвердило коректність роботи REST API, JWT-автентифікації, CRUD-операцій, ownership validation та інтеграції Flutter-клієнта із серверною частиною системи. Отримані результати свідчать про працездатність реалізованої архітектури в межах функціональних вимог бакалаврського проєкту.

3.4 Аналіз результатів роботи системи

Результати реалізації серверної частини мобільного додатку свідчать про коректність обраної архітектури для виконання визначених у підрозділі 1 функціональних вимог бакалаврського проєкту. Використання REST API, модульної структури backend та ORM-підходу дозволило реалізувати взаємодію між Flutter-клієнтом, базою даних та зовнішнім сервісом перекладу у вигляді

незалежних компонентів із чітким розподілом відповідальностей. Подібна архітектура спрощує підтримку системи та полегшує розширення функціоналу без необхідності суттєвих змін у вже реалізованих модулях.

З точки зору software engineering, використання REST API є виправданим для мобільного застосунку, оскільки Flutter-клієнт взаємодіє із сервером через стандартизовані HTTP-запити та JSON-відповіді. Stateless-підхід усуває необхідність зберігання серверних сесій і спрощує масштабування backend-частини. Кожен HTTP-запит містить усю необхідну інформацію для авторизації та обробки даних, що зменшує зв’язність між окремими запитами [54].

Разом із перевагами stateless-архітектура має певні обмеження. Використання JWT спрощує реалізацію авторизації та добре інтегрується з REST API, однак ускладнює механізми logout і дострокового відкликання токенів. Оскільки сервер не зберігає централізовану інформацію про активні сесії, повноцінне керування життєвим циклом токенів у production-системах потребує додаткових механізмів, зокрема refresh token або blacklist token-ів.

Модульна організація backend позитивно вплинула на підтримуваність системи. Виділення окремих router-модулів для авторизації, книг, словника, flashcards та quiz зменшує зв’язність між компонентами й спрощує внесення змін до окремих частин застосунку. Наприклад, модифікація алгоритму інтервального повторення для flashcards не впливає на логіку JWT-автентифікації або CRUD-операції для книг. Подібна ізоляція функціональних компонентів відповідає принципам separation of concerns і є типовою практикою для backend-систем середнього рівня складності.

Важливу роль у структурі серверної частини відіграє dependency injection, реалізований через Depends у FastAPI. Централізоване підключення database session та авторизаційної логіки зменшує дублювання коду між endpoint-ами. Крім цього, подібний підхід спрощує тестування backend-компонентів і робить структуру API більш передбачуваною.

Під час реалізації взаємодії з базою даних використовувався SQLAlchemy ORM. ORM-підхід спростив роботу зі зв’язками, транзакціями та ForeignKey-

залежностями. Порівняно з використанням raw SQL це зробило серверний код більш читабельним і підтримуваним. Оскільки основний акцент робиться на CRUD-операціях і взаємодії між сутностями, можливостей ORM виявилося достатньо. Водночас ORM може створювати додаткові накладні витрати у високонавантажених системах або при виконанні складних аналітичних SQL-запитів, де потрібен повний контроль над структурою запиту.

Структура бази даних відповідає третій нормальній формі. Основні сутності системи винесені в окремі таблиці, а зв'язки між ними реалізовані через ForeignKey-залежності. Подібний підхід зменшує дублювання даних та спрощує підтримку цілісності БД. Наприклад, словникові записи зберігаються окремо від таблиці saved_words, що дозволяє уникнути дублювання однакових перекладів для різних користувачів.

Використання UNIQUE constraints додатково обмежує появу дублікатів. У таблиці reading_progress це гарантує існування лише одного запису прогресу для конкретної пари user_id і book_id, а у dictionary_entries − унікальність словникового запису для комбінації word, source_language та target_language. Подібні обмеження підсилюють узгодженість даних на рівні бази даних і зменшують залежність від перевірок на рівні application logic.

Indexes для frequently queried полів, зокрема user_id, book_id та dictionary_entry_id, використовуються для пришвидшення пошуку та JOIN-операцій. Під час тестування REST API затримок при отриманні книг, закладок або словникових записів не спостерігалося. Водночас збільшення кількості indexes може негативно впливати на швидкість INSERT та UPDATE-операцій, оскільки база даних повинна підтримувати актуальність індексних структур. Тому використання indexes потребує балансу між швидкістю читання та запису даних.

Для підтримки цілісності даних у структурі БД використовується ON DELETE CASCADE. Подібний підхід спрощує видалення пов'язаних сутностей. Наприклад, після видалення користувача автоматично видаляються його книги, закладки, прогрес читання та результати quiz. Це зменшує ризик появи orphan

records у базі даних. Водночас каскадне видалення потребує обережного проєктування зв’язків, оскільки помилкове видалення parent-сутності автоматично впливає на пов’язані записи.

Окремої уваги заслуговує інтеграція із зовнішнім Translation API. Використання стороннього сервісу дозволило реалізувати live-переклад слів без необхідності створення власної системи машинного перекладу. Подібне рішення суттєво спрощує реалізацію функціоналу інтерактивного словника. Проте backend стає залежним від доступності зовнішнього API, стабільності мережевого з’єднання та швидкості відповіді стороннього сервісу. Для часткового зменшення цього ризику у backend використовується timeout для HTTP-запитів через бібліотеку httpx.

Під час інтеграційного тестування Flutter-клієнта та FastAPI-сервера використовувався ngrok, який створював HTTPS-тунель до локального backend-середовища. Це дозволило тестувати мобільний застосунок на фізичному пристрої без окремого production-розгортання. Подібний підхід є зручним на етапі розробки, однак ngrok не може розглядатися як production-рішення через тимчасові URL та обмеження безкоштовної версії сервісу.

Реалізований механізм ownership validation продемонстрував коректний контроль доступу до ресурсів. Використання функції get_owned_book унеможливлює отримання або зміну книг, закладок і прогресу інших користувачів. Подібний контроль авторизації є критично важливим для систем, які працюють із персональними даними користувачів.

Модулі flashcards та quiz демонструють можливість інтеграції навчального функціоналу безпосередньо в систему читання. Використання спрощеного підходу інтервального повторення дозволяє адаптувати інтервал повторення слів залежно від результатів користувача. Разом із цим реалізований алгоритм є базовим і не враховує складніші параметри, характерні для повноцінних систем адаптивного навчання [55].

Проведене функціональне тестування підтвердило коректність роботи основних endpoint-ів, JWT-автентифікації, CRUD-операцій та інтеграції Flutter-

клієнта з серверною частиною. Під час тестування сервер коректно обробляв як позитивні, так і негативні сценарії, включаючи 401 Unauthorized, 404 Not Found та 422 Validation Error. Отримані результати свідчать про працездатність реалізованої архітектури в межах поставлених функціональних вимог.

Подальший розвиток системи може включати реалізацію:

- refresh token-механізму;
- кешування результатів перекладу;
- асинхронну обробку окремих запитів;
- розширення алгоритму інтервального повторення;
- інтеграцію хмарного розгортання backend-частини.

Крім цього, перспективним напрямом є реалізація аналітики навчального прогресу користувача та персоналізованих рекомендацій для повторення лексики.

ВИСНОВКИ

У результаті виконання дипломної роботи спроєктовано та реалізовано серверну частину мобільного застосунку для читання книг із вбудованим інтерактивним словником. У процесі дослідження проаналізовано сучасні підходи до побудови REST-oriented backend-систем, організації взаємодії між мобільним клієнтом і сервером, реалізації механізмів автентифікації та зберігання даних у mobile-oriented applications.

Для реалізації серверної частини обрано технологічний стек на основі FastAPI, MySQL та SQLAlchemy ORM. Використання FastAPI спрощує створення REST API та інтеграцію з Flutter-клієнтом, а також дозволяє централізувати роботу з dependency injection і validation mechanisms. ORM-рівень SQLAlchemy забезпечує абстракцію роботи з базою даних та спрощує підтримку зв’язків між сутностями системи. Разом із цим ORM-підхід створює додатковий abstraction layer між backend-кодом і SQL-запитами, що необхідно враховувати під час подальшого масштабування системи.

У межах роботи спроєктовано структуру бази даних, яка підтримує зберігання інформації про користувачів, книги, прогрес читання, закладки, словникові записи та результати навчальних модулів. Використання ForeignKey-залежностей, зв’язків та механізмів цілісності даних дозволює підтримувати узгодженість між взаємопов’язаними сутностями. Нормалізація структури БД зменшує дублювання даних і спрощує підтримку словникових записів та навчального функціоналу.

Реалізована серверна частина підтримує автентифікацію користувачів, керування бібліотекою книг, механізми збереження прогресу читання та інтеграцію інтерактивного словника з навчальними модулями. Окрему роль у системі відіграє Translation API, через який здійснюється отримання перекладу слів без переходу до сторонніх застосунків. Подібна інтеграція спрощує реалізацію словникового функціоналу, однак створює залежність від зовнішнього сервісу та стабільності його роботи.

Для організації авторизації використано JWT-based authentication. Stateless-підхід добре інтегрується з mobile REST API та спрощує взаємодію між Flutter-клієнтом і серверною частиною без використання server-side sessions. Водночас JWT має характерні обмеження, пов’язані з механізмами token revocation та контролем життєвого циклу токенів.

Архітектура backend-рівня побудована відповідно до принципів modular decomposition. Розподіл функціоналу між окремими router-модулями спрощує підтримку backend-коду та локалізацію змін у системі. Реалізована структура проєкту підтримує подальше розширення функціоналу без суттєвого рефакторингу вже існуючих компонентів.

Під час виконання роботи проведено мануальну верифікацію REST API за допомогою Swagger UI та інтеграційну перевірку взаємодії Flutter-клієнта із FastAPI backend через ngrok. Отримані результати підтверджують коректність реалізації основних функціональних сценаріїв, стабільність обробки HTTP-запитів та працездатність механізмів JWT-автентифікації й ownership validation. Використання ngrok спростило локальну інтеграцію мобільного клієнта та серверної частини, однак подібний підхід не є production-oriented рішенням.

Отримані результати свідчать про коректність реалізації серверної частини мобільного застосунку та узгодженість між архітектурою системи, структурою бази даних і REST API. Обрані технології є достатніми для реалізації функціоналу бакалаврського проєкту та підтримують подальший розвиток системи. Перспективами вдосконалення можуть бути оптимізація продуктивності backend-рівня, розширення механізмів безпеки, інтеграція складніших алгоритмів повторення лексики та розгортання системи у production cloud environment.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Скільки часу люди витрачають на мобільні телефони. BBC News Україна. 2022. URL: https://www.bbc.com/ukrainian/news-59964910 (дата звернення: 20.05.2026).

2. Прасад А. Як читають українці під час вторгнення: частка людей, які читають книги щодня, зросла вдвічі – дослідження УІК. Forbes.ua. 2023. URL: https://forbes.ua/news/yak-chitayut-ukraintsi-pid-chas-vtorgnennya-chastka-lyudey-yaki-chitayut-knigi-shchodnya-zrosla-vdvichi-doslidzhennya-uik-12102023-16629 (дата звернення: 20.05.2026).

3. The pros and cons of e-books. Medium. 2023. URL: https://medium.com/@info_58265/the-pros-and-cons-of-e-books-466525b1e353 (останнє звернення: 20.05.2026).

4. Singer L. M., Alexander P. A. Reading on Paper and Digitally: What the Past Decades of Empirical Research Reveal. Review of Educational Research. 2017. Vol. 87, No. 6. P. 1007–1041.

5. Delgado P., Vargas C., Ackerman R., Salmerón L. Don't throw away your printed books: A meta-analysis on the effects of reading media on reading comprehension. Educational Research Review. 2018. Vol. 25. P. 23–38.

6. Digital reading comprehension instruction in English for children with English as an additional language: A systematic review. Journal of Research in Reading. 2024. Vol. 47, No. 2. P. 115–132. URL: https://www.researchgate.net/publication/384287619_Digital_reading_comprehension_instruction_in_English_for_children_with_English_as_an_additional_language_A_systematic_review (останнє звернення: 20.05.2026).

7. Digital reading comprehension assessment in English language education: a systematic review. Journal of Research in Reading. 2026. Vol. 13, No. 1. URL: https://www.researchgate.net/publication/401048941_Digital_reading_comprehension_assessment_in_English_language_education_a_systematic_review (останнє звернення: 20.05.2026).

8. Noordan M. N. H. B., Yunus M. M. Using Digital Comprehension to Improve Reading Comprehension Skills among Young Learners. International Journal of Academic Research in Progressive Education and Development. 2022. Vol. 11, No. 2. P. 727–748. URL: https://doi.org/10.6007/IJARPED/v11-i2/13208 (останнє звернення: 20.05.2026).

9. Vargo A., Yamaguchi K., Iwata M., Kise K. A Context-Based Multimedia Vocabulary Learning System for Mobile Users. Informatics. 2024. Vol. 11, No. 1. Art. 1. URL: https://doi.org/10.3390/informatics11010001 (останнє звернення: 20.05.2026).

10. Mihaylova M., Gorin S., Reber T. P., Rothen N. A meta-analysis on mobile-assisted language learning applications: Benefits and risks. Frontiers in Psychology. 2022. Vol. 13. Art. 889464.

11. Li K.-C., Sun K.-P., Wong B. T. M., Wu M. M. F. Framework Development for Evaluating the Efficacy of Mobile Language Learning Apps. Electronics. 2025. Vol. 14, No. 8. Art. 1614.

12. Fielding R. T. Architectural Styles and the Design of Network-Based Software Architectures : Doctoral Dissertation. University of California, Irvine, 2000. 162 p.

13. Benchmarking the performance of Python web frameworks. Journal of Computer Sciences Institute. 2025. Vol. 36. P. 336–341. URL: https://www.researchgate.net/publication/396039112_Benchmarking_the_performance_of_Python_web_frameworks (останнє звернення: 20.05.2026).

14. A Performance Evaluation of Flask WSGI and FastAPI ASGI Under High-Concurrency Workloads. Journal of Computer Sciences Institute. 2025. Vol. 36. P. 342–349. URL: https://www.researchgate.net/publication/404177352_A_Performance_Evaluation_of_Flask_WSGI_and_Fastapi_ASGI_Under_High-Concurrency_Workloads (останнє звернення: 20.05.2026).

15. FastAPI. Documentation: https://fastapi.tiangolo.com. 2026. URL: https://fastapi.tiangolo.com (дата звернення: 20.05.2026).

16. Payne R. Beginning App Development with Flutter: Create iOS and Android Apps with Flutter Using Just One Codebase. Berkeley : Apress, 2019. 250 p.

17. ДСТУ ISO/IEC 25010:2016. Інженерія систем і програмних засобів. Вимоги до якості систем і програмних засобів та її оцінювання (SQuaRE). Моделі якості Порталу та програмних засобів (ISO/IEC 25010:2011, IDT). Київ : ДП «УкрНДНЦ», 2016. 47 с.

18. RFC 7519. JSON Web Token (JWT) Specification / Internet Engineering Task Force (IETF). 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (останнє звернення: 20.05.2026).

19. Provos N., Mazières D. A Future-Adaptable Password Scheme. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference. Monterey, 1999. URL: https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf (дата звернення: 20.05.2026).

20. MyMemory API specifications. Translated.net. 2026. URL: https://mymemory.translated.net/doc/spec.php (дата звернення: 20.05.2026).

21. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Ph.D. Dissertation. Irvine : University of California, 2000. 162 p. URL: https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (дата звернення: 15.05.2026).

22. Richardson L., Ruby S. RESTful Web Services. Sebastopol : O'Reilly Media, 2007. 446 p. ISBN 978-0-596-52926-0.

23. Richardson L., Amundsen M., Ruby S. RESTful Web APIs : Services for a Changing World. Sebastopol : O'Reilly Media, 2013. 406 p. ISBN 978-1-449-35806-8.

24. Lubanovic B. FastAPI : Modern Python Web Development. Sebastopol : O'Reilly Media, 2023. 277 p. ISBN 978-1-098-13550-8.

25. Adeshina A. A. Building Python Web APIs with FastAPI : A fast-paced guide to building high-performance, robust web APIs with very little boilerplate code. Birmingham : Packt Publishing, 2022. 214 p. ISBN 978-1-80107-663-0.

26. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley Professional, 2002. 560 p. ISBN 978-0-321-12742-6.

27. SQLAlchemy 2.0 Documentation / SQLAlchemy Authors. 2024. URL: https://docs.sqlalchemy.org/en/20/ (дата звернення: 15.05.2026).

28. MySQL 8.0 Reference Manual / Oracle Corporation. URL: https://dev.mysql.com/doc/refman/8.0/en/ (дата звернення: 15.05.2026).

29. The Twelve-Factor App: III. Config / Wiggins A. URL: https://12factor.net/config (дата звернення: 15.05.2026).

30. Kleppmann M. Designing Data-Intensive Applications : The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 616 p. ISBN 978-1-449-37332-0.

31. Date C. J. An Introduction to Database Systems. 8th ed. Boston : Addison-Wesley, 2003. 1024 p. ISBN 978-0-321-19784-9.

32. Connolly T. M., Begg C. E. Database Systems : A Practical Approach to Design, Implementation, and Management. 6th ed. Boston : Pearson Education, 2014. 1440 p. ISBN 978-0-13-294326-0.

33. DuBois P. MySQL. 5th ed. Boston : Addison-Wesley Professional, 2013. 1153 p. ISBN 978-0-321-83387-7.

34. SQLAlchemy 2.0 Documentation / SQLAlchemy Authors. 2024. URL: https://docs.sqlalchemy.org/en/20/ (дата звернення: 15.05.2026).

35. MySQL 8.0 Reference Manual / Oracle Corporation. URL: https://dev.mysql.com/doc/refman/8.0/en/ (дата звернення: 15.05.2026).

36. ISO/IEC 9075-1:2023. Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). Geneva : International Organization for Standardization, 2023.

37. OpenAPI Specification : Version 3.1.0 / OpenAPI Initiative. 2021. URL: https://spec.openapis.org/oas/v3.1.0 (дата звернення: 15.05.2026).

38. Fielding R., Reschke J. RFC 7231 : Hypertext Transfer Protocol (HTTP/1.1) — Semantics and Content. IETF Standards Track, June 2014. DOI: 10.17487/RFC7231.

39. Jones M., Bradley J., Sakimura N. RFC 7519 : JSON Web Token (JWT). IETF Standards Track, May 2015. DOI: 10.17487/RFC7519. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 15.05.2026).

40. Grassi P. A., Fenton J. L., Newton E. M. et al. NIST Special Publication 800-63B : Digital Identity Guidelines — Authentication and Lifecycle Management. Gaithersburg, MD : National Institute of Standards and Technology, June 2017 (with changes to 02.03.2020). DOI: 10.6028/NIST.SP.800-63b.

41. Jones M., Hardt D. RFC 6750 : The OAuth 2.0 Authorization Framework — Bearer Token Usage. IETF Standards Track, October 2012. DOI: 10.17487/RFC6750. URL: https://datatracker.ietf.org/doc/html/rfc6750 (дата звернення: 15.05.2026).

42. OWASP Password Storage Cheat Sheet / OWASP Foundation. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата звернення: 15.05.2026).

43. OWASP REST Security Cheat Sheet / OWASP Foundation. 2024. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (дата звернення: 15.05.2026).

44. Pydantic Documentation : Version 2.x / Colvin S. et al. 2024. URL: https://docs.pydantic.dev/latest/ (дата звернення: 15.05.2026).

45. Masse M. REST API Design Rulebook : Designing Consistent RESTful Web Service Interfaces. Sebastopol : O'Reilly Media, 2011. 116 p. ISBN 978-1-449-31050-9.

46. HTTPX Documentation : A next-generation HTTP client for Python / Encode OSS Ltd. URL: https://www.python-httpx.org/ (дата звернення: 15.05.2026).

47. Myers J., Copeland R. Essential SQLAlchemy. 2nd ed. Sebastopol : O'Reilly Media, 2016. 234 p. ISBN 978-1-491-91646-9.

48. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley Professional, 2002. 560 p. ISBN 978-0-321-12742-6.

49. Pydantic Documentation : Version 2.x / Colvin S. et al. 2024. URL: https://docs.pydantic.dev/latest/ (дата звернення: 15.05.2026).

50. FastAPI Documentation / Ramírez S. URL: https://fastapi.tiangolo.com/ (дата звернення: 15.05.2026).

51. Lei X., Reynolds B. L. Learning English vocabulary from word cards : A research synthesis. Frontiers in Psychology. 2022. URL: https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2022.984211/full (дата звернення: 15.05.2026).

52. The Twelve-Factor App: III. Config / Wiggins A. URL: https://12factor.net/config (дата звернення: 15.05.2026).

53. Xodabande I., Atai M. R. Using mobile applications for self-directed learning of academic vocabulary among university students. Open Learning : The Journal of Open, Distance and e-Learning. 2022. Vol. 37, No. 4. P. 330–347. DOI: 10.1080/02680513.2020.1847061.

54. ngrok Documentation / ngrok Inc. URL: https://ngrok.com/docs (дата звернення: 15.05.2026).

55. Swagger UI Documentation / SmartBear Software. URL: https://swagger.io/tools/swagger-ui/ (дата звернення: 15.05.2026).

56. Masse M. REST API Design Rulebook : Designing Consistent RESTful Web Service Interfaces. Sebastopol : O'Reilly Media, 2011. 116 p. ISBN 978-1-449-31050-9.

57. Фастовець В. І. Розроблення мобільного програмного забезпечення для вивчення іноземної мови. Вісник Харківського національного автомобільно-дорожнього університету. 2021. № 94. С. 155–163. DOI: 10.30977/BUL.2219-5548.2021.94.0.155.

ДОДАТКИ

ДОДАТОК А

Таблиця А.1 – Основні endpoint-и серверної частини

Модуль	Endpoint	Метод	Призначення	Авторизація
Авторизація	/auth/register	POST	Реєстрація користувача	Ні
Авторизація	/auth/login	POST	Вхід і отримання JWT	Ні
Користувач	/users/me	GET	Отримання поточного користувача	Так
Книги	/books	GET	Отримання списку книг	Так
Книги	/books	POST	Створення книги	Так
Книги	/books/{book_id}	GET	Отримання книги	Так
Книги	/books/{book_id}	PUT	Оновлення книги	Так
Книги	/books/{book_id}	DELETE	Видалення книги	Так
Прогрес	/books/{book_id}/progress	GET	Отримання прогресу	Так
Прогрес	/books/{book_id}/progress	PUT	Оновлення прогресу	Так
Закладки	/books/{book_id}/bookmarks	GET	Отримання закладок	Так
Закладки	/books/{book_id}/bookmarks	POST	Створення закладки	Так

Продовження таблиці А.1

Закладки	/bookmarks/{bookmark_id}	DELETE	Видалення закладки	Так
Переклад	/translate	POST	Отримання перекладу слова	Так
Словник	/dictionary	GET	Отримання збережених слів	Так
Словник	/dictionary	POST	Збереження слова	Так
Словник	/dictionary/{saved_word_id}	DELETE	Видалення слова	Так
Flashcards	/flashcards	GET	Отримання слів для повторення	Так
Flashcards	/flashcards/{id}	PUT	Оновлення результату повторення	Так
Quiz	/quiz/question	GET	Генерація питання	Так
Quiz	/quiz/answer	POST	Перевірка відповіді	Так

ДОДАТОК Б

Фрагменти backend-коду серверної частини

Лістинг Б.1 – Ініціалізація FastAPI-застосунку та підключення router-модулів (main.py)

```
from contextlib import asynccontextmanager

from fastapi import FastAPI

from app.database import Base, engine
from app.routes import auth, bookmarks, books, dictionary, flashcards, progress, quiz, translate, users


@asynccontextmanager
async def lifespan(app: FastAPI):
    Base.metadata.create_all(bind=engine)
    yield


app = FastAPI(
    title="Reading App API",
    description="Standalone Reading App backend with FastAPI, SQLAlchemy, MySQL, JWT, and bcrypt.",
    version="1.0.0",
    lifespan=lifespan,
)

app.include_router(auth.router)
app.include_router(users.router)
app.include_router(books.router)
app.include_router(progress.router)
app.include_router(bookmarks.router)
app.include_router(dictionary.router)
app.include_router(flashcards.router)
app.include_router(translate.router)
```

```
app.include_router(quiz.router)


@app.get("/health")
def health_check():
    return {"status": "ok"}
```

Лістинг Б.2 – Підключення до бази даних та реалізація dependency injection (database.py)

```
import os

from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase, sessionmaker

load_dotenv()

DATABASE_URL = os.getenv(
    "DATABASE_URL",

"mysql+pymysql://root:UriahHeep1970%21@localhost:3306/reading_app"
,
)

engine = create_engine(
    DATABASE_URL,
    pool_pre_ping=True,
    pool_recycle=3600,
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)


class Base(DeclarativeBase):
```

```
    pass


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Лістинг Б.3 – Реалізація JWT-автентифікації та хешування паролів (app/auth.py)

```
import os
from datetime import datetime, timedelta, timezone
from typing import Annotated

from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPAuthorizationCredentials,
HTTPBearer
from jose import JWTError, jwt
from passlib.context import CryptContext
from sqlalchemy.orm import Session

from app import models
from app.database import get_db

SECRET_KEY = os.getenv("JWT_SECRET_KEY", "change-this-secret-in-
production")
ALGORITHM = os.getenv("JWT_ALGORITHM", "HS256")
ACCESS_TOKEN_EXPIRE_MINUTES =
int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "60"))

bearer_scheme = HTTPBearer()
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

```
def hash_password(password: str) -> str:
    password = str(password)
    if len(password.encode("utf-8")) > 72:
        password = password.encode("utf-8")[:72].decode("utf-8",
errors="ignore")
    return pwd_context.hash(password)

def verify_password(plain_password: str, password_hash: str) ->
bool:
    plain_password = str(plain_password)
    if len(plain_password.encode("utf-8")) > 72:
        plain_password = plain_password.encode("utf-
8")[:72].decode("utf-8", errors="ignore")
    return pwd_context.verify(plain_password, password_hash)


def create_access_token(subject: str, expires_delta: timedelta |
None = None) -> str:
    expire = datetime.now(timezone.utc) + (expires_delta or
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    payload = {"sub": subject, "exp": expire}
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)


def authenticate_user(db: Session, email: str, password: str) ->
models.User | None:
    user = db.query(models.User).filter(models.User.email ==
email).first()
    if not user or not verify_password(password,
user.password_hash):
        return None
    return user
```

```
def get_current_user(
    credentials:            Annotated[HTTPAuthorizationCredentials,
Depends(bearer_scheme)],
    db: Annotated[Session, Depends(get_db)],
) -> models.User:
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    token = credentials.credentials

    try:
        payload       =       jwt.decode(token,        SECRET_KEY,
algorithms=[ALGORITHM])
        subject = payload.get("sub")
        if subject is None:
            raise credentials_exception
        user_id = int(subject)
    except (JWTError, ValueError):
        raise credentials_exception

    user = db.get(models.User, user_id)
    if user is None:
        raise credentials_exception
    return user
```

Лістинг Б.4 – ORM-моделі та relationships серверної частини (models.py)

```
from datetime import datetime, timezone

from sqlalchemy import Boolean, DateTime, ForeignKey, Index,
Integer, String, Text, UniqueConstraint
from sqlalchemy.orm import Mapped, mapped_column, relationship
```

```
from sqlalchemy import Enum
from sqlalchemy import Text
from sqlalchemy import Float

from app.database import Base


def utcnow() -> datetime:
    return datetime.now(timezone.utc)


class User(Base):
    __tablename__ = "users"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
index=True)
    email: Mapped[str] = mapped_column(String(255), unique=True,
index=True, nullable=False)
    username: Mapped[str] = mapped_column(String(80), unique=True,
index=True, nullable=False)
    password_hash: Mapped[str] = mapped_column(String(255),
nullable=False)
    created_at: Mapped[datetime] =
mapped_column(DateTime(timezone=True), default=utcnow,
nullable=False)
    updated_at: Mapped[datetime] =
mapped_column(DateTime(timezone=True), default=utcnow,
onupdate=utcnow, nullable=False)

    books: Mapped[list["Book"]] =
relationship(back_populates="owner", cascade="all, delete-orphan")
    reading_progress: Mapped[list["ReadingProgress"]] =
relationship(back_populates="user", cascade="all, delete-orphan")
    bookmarks: Mapped[list["Bookmark"]] =
relationship(back_populates="user", cascade="all, delete-orphan")
```

```
    saved_words:            Mapped[list["SavedWord"]]           =
relationship(back_populates="user", cascade="all, delete-orphan")
    quiz_attempts:          Mapped[list["QuizAttempt"]]         =
relationship(back_populates="user", cascade="all, delete-orphan")


class Book(Base):
    __tablename__ = "books"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
index=True)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id",
ondelete="CASCADE"), index=True, nullable=False)
    title: Mapped[str] = mapped_column(String(255), nullable=False)
    author: Mapped[str | None] = mapped_column(String(255),
nullable=True)
    description: Mapped[str | None] = mapped_column(Text,
nullable=True)
    content: Mapped[str] = mapped_column(Text, nullable=False)
    language: Mapped[str] = mapped_column(String(50),
nullable=False)
    cover_url: Mapped[str | None] = mapped_column(String(1024),
nullable=True)
    genre: Mapped[str | None] = mapped_column(String(120),
nullable=True)
    status: Mapped[str] = mapped_column(
        Enum(
        "not_started",
        "reading",
        "completed",
        "archived",
        name="book_status_enum"
        ),
        default="not_started",
        nullable=False
```

```
        )
    created_at:                    Mapped[datetime]                    =
mapped_column(DateTime(timezone=True),              default=utcnow,
nullable=False)
    updated_at:                    Mapped[datetime]                    =
mapped_column(DateTime(timezone=True),              default=utcnow,
onupdate=utcnow, nullable=False)
    last_opened_at:      Mapped[datetime      |      None]      =
mapped_column(DateTime(timezone=True), nullable=True)

    owner: Mapped[User] = relationship(back_populates="books")
    progress:      Mapped["ReadingProgress      |      None"]      =
relationship(back_populates="book", cascade="all, delete-orphan")
    bookmarks:              Mapped[list["Bookmark"]]              =
relationship(back_populates="book", cascade="all, delete-orphan")
    saved_words:             Mapped[list["SavedWord"]]            =
relationship(back_populates="source_book")

    __table_args__ = (
        Index("ix_books_user_created", "user_id", "created_at"),
    )


class ReadingProgress(Base):
    __tablename__ = "reading_progress"

    id:  Mapped[int]  =  mapped_column(Integer,  primary_key=True,
index=True)
    user_id:  Mapped[int]  =  mapped_column(ForeignKey("users.id",
ondelete="CASCADE"), index=True, nullable=False)
    book_id:  Mapped[int]  =  mapped_column(ForeignKey("books.id",
ondelete="CASCADE"), index=True, nullable=False)
    paragraph_index:    Mapped[int]    =    mapped_column(Integer,
default=0, nullable=False)
```

```
    character_index:    Mapped[int]    =    mapped_column(Integer,
default=0, nullable=False)
    created_at:                  Mapped[datetime]                  =
mapped_column(DateTime(timezone=True),              default=utcnow,
nullable=False)
    updated_at:                  Mapped[datetime]                  =
mapped_column(DateTime(timezone=True),              default=utcnow,
onupdate=utcnow, nullable=False)

    user:                        Mapped[User]                      =
relationship(back_populates="reading_progress")
    book: Mapped[Book] = relationship(back_populates="progress")

    __table_args__ = (
        UniqueConstraint("user_id",                        "book_id",
name="uq_reading_progress_user_book"),
    )


class Bookmark(Base):
    __tablename__ = "bookmarks"

    id:  Mapped[int]  =  mapped_column(Integer,  primary_key=True,
index=True)
    user_id:  Mapped[int]  =  mapped_column(ForeignKey("users.id",
ondelete="CASCADE"), index=True, nullable=False)
    book_id:  Mapped[int]  =  mapped_column(ForeignKey("books.id",
ondelete="CASCADE"), index=True, nullable=False)
    paragraph_index:    Mapped[int]    =    mapped_column(Integer,
nullable=False)
    character_index:    Mapped[int]    =    mapped_column(Integer,
nullable=False)
    note: Mapped[str | None] = mapped_column(Text, nullable=True)
```

```
    created_at:                 Mapped[datetime]                =
mapped_column(DateTime(timezone=True),            default=utcnow,
nullable=False)
    updated_at:                 Mapped[datetime]                =
mapped_column(DateTime(timezone=True),            default=utcnow,
onupdate=utcnow, nullable=False)
    selected_text:  Mapped[str  |  None]  =  mapped_column(Text,
nullable=True)

    user: Mapped[User] = relationship(back_populates="bookmarks")
    book: Mapped[Book] = relationship(back_populates="bookmarks")

    __table_args__ = (
        Index("ix_bookmarks_user_book", "user_id", "book_id"),
    )


class DictionaryEntry(Base):
    __tablename__ = "dictionary_entries"

    id:  Mapped[int]  =  mapped_column(Integer,  primary_key=True,
index=True)
    word: Mapped[str] = mapped_column(String(255), nullable=False)
    translation:    Mapped[str]    =    mapped_column(String(255),
nullable=False)
    source_language:   Mapped[str]   =   mapped_column(String(50),
nullable=False)
    target_language:   Mapped[str]   =   mapped_column(String(50),
nullable=False)
    created_at:                 Mapped[datetime]                =
mapped_column(DateTime(timezone=True),            default=utcnow,
nullable=False)
```

```
    saved_words: Mapped[list["SavedWord"]] = relationship(back_populates="dictionary_entry", cascade="all, delete-orphan")

    __table_args__ = (
        UniqueConstraint(
            "word",
            "source_language",
            "target_language",
            name="uq_dictionary_entry_text_languages",
        ),
    )


class SavedWord(Base):
    __tablename__ = "saved_words"

    id: Mapped[int] = mapped_column(Integer, primary_key=True, index=True)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id", ondelete="CASCADE"), index=True, nullable=False)
    dictionary_entry_id: Mapped[int] = mapped_column(ForeignKey("dictionary_entries.id", ondelete="CASCADE"), index=True, nullable=False)
    source_book_id: Mapped[int | None] = mapped_column(ForeignKey("books.id", ondelete="SET NULL"), nullable=True)
    difficulty: Mapped[str] = mapped_column(String(20), default="medium", nullable=False)
    last_reviewed_at: Mapped[datetime | None] = mapped_column(DateTime(timezone=True), nullable=True)
    next_review_at: Mapped[datetime | None] = mapped_column(DateTime(timezone=True), nullable=True)
```

```
    saved_at:                    Mapped[datetime]                    =
mapped_column(DateTime(timezone=True),              default=utcnow,
nullable=False)
    updated_at:                  Mapped[datetime]                    =
mapped_column(DateTime(timezone=True),              default=utcnow,
onupdate=utcnow, nullable=False)

    user: Mapped[User] = relationship(back_populates="saved_words")
    dictionary_entry:          Mapped[DictionaryEntry]           =
relationship(back_populates="saved_words")
    source_book:        Mapped[Book        |        None]        =
relationship(back_populates="saved_words")
    quiz_attempts:           Mapped[list["QuizAttempt"]]         =
relationship(back_populates="saved_word",   cascade="all,   delete-
orphan")

    __table_args__ = (
        UniqueConstraint("user_id",          "dictionary_entry_id",
name="uq_saved_word_user_entry"),
        Index("ix_saved_words_user_next_review",         "user_id",
"next_review_at"),
    )


class QuizAttempt(Base):
    __tablename__ = "quiz_attempts"

    id:  Mapped[int]  =  mapped_column(Integer,  primary_key=True,
index=True)
    user_id:  Mapped[int]  =  mapped_column(ForeignKey("users.id",
ondelete="CASCADE"), index=True, nullable=False)
    saved_word_id:                Mapped[int]                       =
mapped_column(ForeignKey("saved_words.id",     ondelete="CASCADE"),
index=True, nullable=False)
```

```
    selected_answer: Mapped[str] = mapped_column(String(255),
nullable=False)
    correct_answer: Mapped[str] = mapped_column(String(255),
nullable=False)
    is_correct: Mapped[bool] = mapped_column(Boolean,
nullable=False)
    created_at: Mapped[datetime] =
mapped_column(DateTime(timezone=True), default=utcnow,
nullable=False)

    user: Mapped[User] =
relationship(back_populates="quiz_attempts")
    saved_word: Mapped[SavedWord] =
relationship(back_populates="quiz_attempts")

    __table_args__ = (
        Index("ix_quiz_attempts_user_created", "user_id",
"created_at"),
        Index("ix_quiz_attempts_user_saved_word", "user_id",
"saved_word_id"),
    )
```

Лістинг Б.5 – Pydantic DTO-схеми для валідації запитів і відповідей (schemas.py)

```
from datetime import datetime
from typing import Literal

from pydantic import BaseModel, ConfigDict, EmailStr, Field


BookStatus = Literal["not_started", "reading", "completed",
"archived"]
Difficulty = Literal["easy", "medium", "hard"]
```

```
class Token(BaseModel):
    access_token: str
    token_type: str = "bearer"


class UserRegister(BaseModel):
    email: EmailStr
    username: str = Field(min_length=3, max_length=80)
    password: str = Field(min_length=8, max_length=128)


class UserLogin(BaseModel):
    email: EmailStr
    password: str


class UserRead(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: int
    email: EmailStr
    username: str
    created_at: datetime
    updated_at: datetime


class BookBase(BaseModel):
    title: str = Field(min_length=1, max_length=255)
    author: str | None = Field(default=None, max_length=255)
    description: str | None = None
    content: str = Field(min_length=1)
    language: str = Field(min_length=1, max_length=50)
    cover_url: str | None = Field(default=None, max_length=1024)
    genre: str | None = Field(default=None, max_length=120)
    status: BookStatus = "not_started"
```

```
    last_opened_at: datetime | None = None


class BookCreate(BookBase):
    pass


class BookUpdate(BaseModel):
    title: str | None = Field(default=None, min_length=1,
max_length=255)
    author: str | None = Field(default=None, max_length=255)
    description: str | None = None
    content: str | None = Field(default=None, min_length=1)
    language: str | None = Field(default=None, min_length=1,
max_length=50)
    cover_url: str | None = Field(default=None, max_length=1024)
    genre: str | None = Field(default=None, max_length=120)
    status: BookStatus | None = None
    last_opened_at: datetime | None = None


class BookRead(BookBase):
    model_config = ConfigDict(from_attributes=True)

    id: int
    user_id: int
    created_at: datetime
    updated_at: datetime


class ProgressBase(BaseModel):
    paragraph_index: int = Field(ge=0)
    character_index: int = Field(ge=0)
```

```
class ProgressUpdate(ProgressBase):
    pass


class ProgressRead(ProgressBase):
    model_config = ConfigDict(from_attributes=True)

    id: int
    user_id: int
    book_id: int
    created_at: datetime
    updated_at: datetime


class BookmarkCreate(BaseModel):
    paragraph_index: int = Field(ge=0)
    character_index: int = Field(ge=0)
    selected_text: str | None = None
    note: str | None = None


class BookmarkRead(BookmarkCreate):
    model_config = ConfigDict(from_attributes=True)

    id: int
    user_id: int
    book_id: int
    selected_text: str | None = None
    created_at: datetime
    updated_at: datetime


class DictionaryCreate(BaseModel):
    word: str = Field(min_length=1, max_length=255)
    translation: str = Field(min_length=1, max_length=255)
```

```
    source_language: str = Field(min_length=1, max_length=50)
    target_language: str = Field(min_length=1, max_length=50)
    source_book_id: int | None = None
    difficulty: Difficulty = "medium"


class DictionaryEntryRead(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: int
    word: str
    translation: str
    source_language: str
    target_language: str
    created_at: datetime


class SavedWordRead(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: int
    user_id: int
    dictionary_entry_id: int
    source_book_id: int | None
    difficulty: Difficulty
    last_reviewed_at: datetime | None
    next_review_at: datetime | None
    saved_at: datetime
    updated_at: datetime
    dictionary_entry: DictionaryEntryRead


class FlashcardRead(BaseModel):
    id: int
    saved_word_id: int
```

```
    word: str
    translation: str
    source_language: str
    target_language: str
    difficulty: Difficulty
    last_reviewed_at: datetime | None
    next_review_at: datetime | None


class FlashcardUpdate(BaseModel):
    correct: bool
    difficulty: Difficulty | None = None


class TranslationRequest(BaseModel):
    word: str = Field(min_length=1, max_length=255)
    source_language: str = Field(min_length=1, max_length=50)
    target_language: str = Field(min_length=1, max_length=50)


class TranslationResponse(BaseModel):
    word: str
    translation: str


class QuizQuestionRead(BaseModel):
    question_id: str
    word: str
    options: list[str]


class QuizAnswerCreate(BaseModel):
    question_id: str
    selected_answer: str = Field(min_length=1, max_length=255)
```

```
class QuizAnswerRead(BaseModel):
    is_correct: bool


class QuizAttemptRead(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: int
    user_id: int
    saved_word_id: int
    selected_answer: str
    correct_answer: str
    is_correct: bool
    created_at: datetime
```

Лістинг Б.6 – Реалізація endpoint-ів автентифікації користувачів (routes/auth.py)

```
from datetime import timedelta

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.exc import IntegrityError
from sqlalchemy.orm import Session

from app import auth as auth_service, models, schemas
from app.database import get_db

router = APIRouter(prefix="/auth", tags=["auth"])


@router.post("/register", response_model=schemas.UserRead,
status_code=status.HTTP_201_CREATED)
def register(payload: schemas.UserRegister, db: Session =
Depends(get_db)):
```

```
    existing = (
        db.query(models.User)
        .filter((models.User.email == payload.email) |
(models.User.username == payload.username))
        .first()
    )
    if existing:
        raise HTTPException(status_code=409, detail="Email or
username already exists")

    user = models.User(
        email=payload.email,
        username=payload.username,

password_hash=auth_service.hash_password(payload.password),
    )
    db.add(user)
    try:
        db.commit()
    except IntegrityError:
        db.rollback()
        raise HTTPException(status_code=409, detail="Email or
username already exists")

    db.refresh(user)
    return user


@router.post("/login", response_model=schemas.Token)
def login(payload: schemas.UserLogin, db: Session =
Depends(get_db)):
    user = auth_service.authenticate_user(db, payload.email,
payload.password)
    if not user:
        raise HTTPException(
```

```
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid email or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    token = auth_service.create_access_token(
        subject=str(user.id),

expires_delta=timedelta(minutes=auth_service.ACCESS_TOKEN_EXPIRE_M
INUTES),
    )
    return schemas.Token(access_token=token)
```

Лістинг Б.7 – Реалізація endpoint-а отримання поточного користувача

```
users.py
from fastapi import APIRouter, Depends

from app import models, schemas
from app.auth import get_current_user

router = APIRouter(prefix="/users", tags=["users"])


@router.get("/me", response_model=schemas.UserRead)
def get_me(current_user: models.User = Depends(get_current_user)):
    return current_user
```

Лістинг Б.8 – Реалізація ownership validation для перевірки доступу до книги (books.py)

```
from fastapi import APIRouter, Depends, HTTPException, Response,
status
from sqlalchemy.orm import Session

from app import models, schemas
from app.auth import get_current_user
```

```
from app.database import get_db

router = APIRouter(prefix="/books", tags=["books"])


def get_owned_book(db: Session, user_id: int, book_id: int) ->
models.Book:
    book = db.query(models.Book).filter(models.Book.id == book_id,
models.Book.user_id == user_id).first()
    if not book:
        raise HTTPException(status_code=404, detail="Book not
found")
    return book


@router.get("", response_model=list[schemas.BookRead])
def list_books(
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    return (
        db.query(models.Book)
        .filter(models.Book.user_id == current_user.id)
        .order_by(models.Book.created_at.desc())
        .all()
    )


@router.post("", response_model=schemas.BookRead,
status_code=status.HTTP_201_CREATED)
def create_book(
    payload: schemas.BookCreate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
```

```
    book = models.Book(user_id=current_user.id,
**payload.model_dump())
    db.add(book)
    db.commit()
    db.refresh(book)
    return book


@router.get("/{book_id}", response_model=schemas.BookRead)
def get_book(
    book_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    return get_owned_book(db, current_user.id, book_id)


@router.put("/{book_id}", response_model=schemas.BookRead)
def update_book(
    book_id: int,
    payload: schemas.BookUpdate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    book = get_owned_book(db, current_user.id, book_id)
    for field, value in
payload.model_dump(exclude_unset=True).items():
        setattr(book, field, value)
    db.commit()
    db.refresh(book)
    return book


@router.delete("/{book_id}",
status_code=status.HTTP_204_NO_CONTENT)
```

```
def delete_book(
    book_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    book = get_owned_book(db, current_user.id, book_id)
    db.delete(book)
    db.commit()
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Лістинг Б.9 – Реалізація flashcards-модуля (flashcards.py)

```
from datetime import timedelta

from fastapi import APIRouter, Depends, HTTPException, Query
from sqlalchemy import or_
from sqlalchemy.orm import Session, joinedload

from app import models, schemas
from app.auth import get_current_user
from app.database import get_db
from app.models import utcnow

router = APIRouter(prefix="/flashcards", tags=["flashcards"])


def to_flashcard(saved_word: models.SavedWord) ->
schemas.FlashcardRead:
    entry = saved_word.dictionary_entry
    return schemas.FlashcardRead(
        id=saved_word.id,
        saved_word_id=saved_word.id,
        word=entry.word,
        translation=entry.translation,
        source_language=entry.source_language,
        target_language=entry.target_language,
```

```
            difficulty=saved_word.difficulty,
            last_reviewed_at=saved_word.last_reviewed_at,
            next_review_at=saved_word.next_review_at,
        )


def next_review_interval(correct: bool, difficulty: str) ->
timedelta:
    if not correct:
        return timedelta(minutes=15)
    if difficulty == "easy":
        return timedelta(days=7)
    if difficulty == "medium":
        return timedelta(days=3)
    return timedelta(days=1)


@router.get("", response_model=list[schemas.FlashcardRead])
def get_flashcards(
    limit: int = Query(default=20, ge=1, le=100),
    due_only: bool = True,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    now = utcnow()
    query = (
        db.query(models.SavedWord)
        .options(joinedload(models.SavedWord.dictionary_entry))
        .filter(models.SavedWord.user_id == current_user.id)
    )
    if due_only:
        query                                                    =
query.filter(or_(models.SavedWord.next_review_at.is_(None),
models.SavedWord.next_review_at <= now))
```

```
    saved_words =
query.order_by(models.SavedWord.next_review_at.asc(),
models.SavedWord.saved_at.asc()).limit(limit).all()
    return [to_flashcard(saved_word) for saved_word in saved_words]


@router.put("/{id}", response_model=schemas.FlashcardRead)
def update_flashcard(
    id: int,
    payload: schemas.FlashcardUpdate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    saved_word = (
        db.query(models.SavedWord)
        .options(joinedload(models.SavedWord.dictionary_entry))
        .filter(models.SavedWord.id == id, models.SavedWord.user_id
== current_user.id)
        .first()
    )
    if not saved_word:
        raise HTTPException(status_code=404, detail="Flashcard not
found")

    if payload.difficulty is not None:
        saved_word.difficulty = payload.difficulty
    elif not payload.correct:
        saved_word.difficulty = "hard"

    now = utcnow()
    saved_word.last_reviewed_at = now
    saved_word.next_review_at = now +
next_review_interval(payload.correct, saved_word.difficulty)
    db.commit()
    db.refresh(saved_word)
```

```
    return to_flashcard(saved_word)
```

Лістинг Б.10 – Реалізація збереження та отримання словникових записів (dictionary.py)

```
from fastapi import APIRouter, Depends, HTTPException, Response, status
from sqlalchemy.orm import Session, joinedload

from app import models, schemas
from app.auth import get_current_user
from app.database import get_db
from app.routes.books import get_owned_book

router = APIRouter(prefix="/dictionary", tags=["dictionary"])


@router.get("", response_model=list[schemas.SavedWordRead])
def list_dictionary(
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    return (
        db.query(models.SavedWord)
        .options(joinedload(models.SavedWord.dictionary_entry))
        .filter(models.SavedWord.user_id == current_user.id)
        .order_by(models.SavedWord.saved_at.desc())
        .all()
    )


@router.post("",                response_model=schemas.SavedWordRead,
status_code=status.HTTP_201_CREATED)
def save_word(
    payload: schemas.DictionaryCreate,
    db: Session = Depends(get_db),
```

```
    current_user: models.User = Depends(get_current_user),
):
    if payload.source_book_id is not None:
        get_owned_book(db, current_user.id, payload.source_book_id)

    entry = (
        db.query(models.DictionaryEntry)
        .filter(
            models.DictionaryEntry.word == payload.word,
            models.DictionaryEntry.source_language == payload.source_language,
            models.DictionaryEntry.target_language == payload.target_language,
        )
        .first()
    )
    if not entry:
        entry = models.DictionaryEntry(
            word=payload.word,
            translation=payload.translation,
            source_language=payload.source_language,
            target_language=payload.target_language,
        )
        db.add(entry)
        db.flush()

    saved_word = (
        db.query(models.SavedWord)
        .filter(models.SavedWord.user_id == current_user.id, models.SavedWord.dictionary_entry_id == entry.id)
        .first()
    )
    if saved_word:
        saved_word.source_book_id = payload.source_book_id or saved_word.source_book_id
```

```
            saved_word.difficulty = payload.difficulty
        else:
            saved_word = models.SavedWord(
                user_id=current_user.id,
                dictionary_entry_id=entry.id,
                source_book_id=payload.source_book_id,
                difficulty=payload.difficulty,
            )
            db.add(saved_word)

        db.commit()
        return (
            db.query(models.SavedWord)
            .options(joinedload(models.SavedWord.dictionary_entry))
            .filter(models.SavedWord.id == saved_word.id,
models.SavedWord.user_id == current_user.id)
            .one()
        )


@router.delete("/{saved_word_id}",
status_code=status.HTTP_204_NO_CONTENT)
def delete_saved_word(
    saved_word_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    saved_word = (
        db.query(models.SavedWord)
        .filter(models.SavedWord.id == saved_word_id,
models.SavedWord.user_id == current_user.id)
        .first()
    )
    if not saved_word:
```

```
        raise HTTPException(status_code=404, detail="Saved word not
found")

    db.delete(saved_word)
    db.commit()
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Лістинг Б.11 – Реалізація збереження прогресу читання (progress.py)

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session

from app import models, schemas
from app.auth import get_current_user
from app.database import get_db
from app.routes.books import get_owned_book

router = APIRouter(prefix="/books/{book_id}/progress",
tags=["reading progress"])


@router.get("", response_model=schemas.ProgressRead)
def get_progress(
    book_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    get_owned_book(db, current_user.id, book_id)
    progress = (
        db.query(models.ReadingProgress)
        .filter(models.ReadingProgress.user_id == current_user.id,
models.ReadingProgress.book_id == book_id)
        .first()
    )
    if progress:
        return progress
```

```
    progress = models.ReadingProgress(user_id=current_user.id,
book_id=book_id)
    db.add(progress)
    db.commit()
    db.refresh(progress)
    return progress


@router.put("", response_model=schemas.ProgressRead)
def update_progress(
    book_id: int,
    payload: schemas.ProgressUpdate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    book = get_owned_book(db, current_user.id, book_id)
    progress = (
        db.query(models.ReadingProgress)
        .filter(models.ReadingProgress.user_id == current_user.id,
models.ReadingProgress.book_id == book.id)
        .first()
    )
    if not progress:
        progress = models.ReadingProgress(user_id=current_user.id,
book_id=book.id)
        db.add(progress)

    progress.paragraph_index = payload.paragraph_index
    progress.character_index = payload.character_index
    db.commit()
    db.refresh(progress)
    return progress
```

Лістинг Б.12 – Реалізація роботи із закладками (bookmarks.py)

```
from fastapi import APIRouter, Depends, HTTPException, Response,
status
from sqlalchemy.orm import Session

from app import models, schemas
from app.auth import get_current_user
from app.database import get_db
from app.routes.books import get_owned_book

router = APIRouter(tags=["bookmarks"])


@router.get("/books/{book_id}/bookmarks",
response_model=list[schemas.BookmarkRead])
def list_bookmarks(
    book_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    get_owned_book(db, current_user.id, book_id)
    return (
        db.query(models.Bookmark)
        .filter(models.Bookmark.user_id     ==     current_user.id,
models.Bookmark.book_id == book_id)
        .order_by(models.Bookmark.created_at.desc())
        .all()
    )


@router.post("/books/{book_id}/bookmarks",
response_model=schemas.BookmarkRead,
status_code=status.HTTP_201_CREATED)
def add_bookmark(
    book_id: int,
    payload: schemas.BookmarkCreate,
```

```
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    get_owned_book(db, current_user.id, book_id)
    bookmark = models.Bookmark(user_id=current_user.id,
book_id=book_id, **payload.model_dump())
    db.add(bookmark)
    db.commit()
    db.refresh(bookmark)
    return bookmark


@router.delete("/bookmarks/{bookmark_id}",
status_code=status.HTTP_204_NO_CONTENT)
def delete_bookmark(
    bookmark_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    bookmark = (
        db.query(models.Bookmark)
        .filter(models.Bookmark.id == bookmark_id,
models.Bookmark.user_id == current_user.id)
        .first()
    )
    if not bookmark:
        raise HTTPException(status_code=404, detail="Bookmark not
found")

    db.delete(bookmark)
    db.commit()
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Лістинг Б.13 – Реалізація quiz-модуля (quiz.py)

```
router = APIRouter(prefix="/quiz", tags=["quiz"])
```

```
QUIZ_TOKEN_PREFIX = "quiz"
QUIZ_TOKEN_EXPIRE_MINUTES = 15
MIN_OPTIONS = 2
MAX_OPTIONS = 4


def create_question_id(user_id: int, saved_word_id: int) -> str:
    return auth_service.create_access_token(
        subject=f"{QUIZ_TOKEN_PREFIX}:{user_id}:{saved_word_id}",

expires_delta=timedelta(minutes=QUIZ_TOKEN_EXPIRE_MINUTES),
    )


def parse_question_id(question_id: str, current_user_id: int) ->
int:
    credentials_exception = HTTPException(
        status_code=status.HTTP_400_BAD_REQUEST,
        detail="Invalid or expired question_id",
    )
    try:
        payload = jwt.decode(question_id, auth_service.SECRET_KEY,
algorithms=[auth_service.ALGORITHM])
        subject = payload.get("sub")
        if not isinstance(subject, str):
            raise credentials_exception
        prefix, user_id_text, saved_word_id_text =
subject.split(":", 2)
        if prefix != QUIZ_TOKEN_PREFIX or int(user_id_text) !=
current_user_id:
            raise credentials_exception
        return int(saved_word_id_text)
    except (JWTError, ValueError):
        raise credentials_exception
```

```
def get_user_saved_word(db: Session, user_id: int, saved_word_id:
int) -> models.SavedWord:
    saved_word = (
        db.query(models.SavedWord)
        .options(joinedload(models.SavedWord.dictionary_entry))
        .filter(models.SavedWord.id == saved_word_id,
models.SavedWord.user_id == user_id)
        .first()
    )
    if not saved_word:
        raise HTTPException(status_code=404, detail="Quiz question
not found")
    return saved_word


@router.get("/question", response_model=schemas.QuizQuestionRead)
def get_quiz_question(
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    saved_words = (
        db.query(models.SavedWord)
        .options(joinedload(models.SavedWord.dictionary_entry))
        .filter(models.SavedWord.user_id == current_user.id)
        .all()
    )
    if not saved_words:
        raise HTTPException(status_code=404, detail="No saved words
available for quiz")

    question_word = random.choice(saved_words)
    correct_answer = question_word.dictionary_entry.translation
```

```
    distractors = [
        saved_word.dictionary_entry.translation
        for saved_word in saved_words
        if saved_word.id != question_word.id and
saved_word.dictionary_entry.translation != correct_answer
    ]
    random.shuffle(distractors)

    fallback_options = ["translation not available", "I do not
know", "skip"]
    options = [correct_answer, *distractors[: MAX_OPTIONS - 1]]
    options = list(dict.fromkeys(options))
    for fallback in fallback_options:
        if len(options) >= MIN_OPTIONS:
            break
        if fallback != correct_answer and fallback not in options:
            options.append(fallback)
    random.shuffle(options)

    return schemas.QuizQuestionRead(
        question_id=create_question_id(current_user.id,
question_word.id),
        word=question_word.dictionary_entry.word,
        options=options,
    )


@router.post("/answer", response_model=schemas.QuizAnswerRead)
def answer_quiz_question(
    payload: schemas.QuizAnswerCreate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
):
    saved_word_id = parse_question_id(payload.question_id,
current_user.id)
```

```
    saved_word = get_user_saved_word(db, current_user.id,
saved_word_id)
    correct_answer = saved_word.dictionary_entry.translation
    is_correct = payload.selected_answer.strip() ==
correct_answer.strip()

    attempt = models.QuizAttempt(
        user_id=current_user.id,
        saved_word_id=saved_word.id,
        selected_answer=payload.selected_answer,
        correct_answer=correct_answer,
        is_correct=is_correct,
    )
    db.add(attempt)
    db.commit()

    return schemas.QuizAnswerRead(is_correct=is_correct)
```

Лістинг Б.14 – Реалізація endpoint-а перекладу слова (translate.py)

```
from fastapi import APIRouter, Depends, HTTPException, status

from app import models, schemas
from app.auth import get_current_user
from app.translation import translate_word

router = APIRouter(prefix="/translate", tags=["translation"])


@router.post("", response_model=schemas.TranslationResponse)
def translate(
    payload: schemas.TranslationRequest,
    current_user: models.User = Depends(get_current_user),
):
    try:
        translated_word = translate_word(
```

```
            word=payload.word,
            source_language=payload.source_language,
            target_language=payload.target_language,
        )
    except ValueError as exc:
        raise
HTTPException(status_code=status.HTTP_502_BAD_GATEWAY,
detail=str(exc))

    return schemas.TranslationResponse(
        word=payload.word,
        translation=translated_word,
    )
```

Лістинг Б.15 – Реалізація інтеграції із зовнішнім Translation API

translation.py

```
import os

import httpx
from dotenv import load_dotenv

load_dotenv()

TRANSLATION_API_URL = os.getenv(
    "TRANSLATION_API_URL",
    "https://api.mymemory.translated.net/get"
)
TRANSLATION_TIMEOUT_SECONDS                                    =
float(os.getenv("TRANSLATION_TIMEOUT_SECONDS", "10"))
TRANSLATION_NOT_AVAILABLE = "translation not available"


def translate_word(word: str, source_language: str, target_language:
str) -> str:
    params = {
```

```
        "q": word,
        "langpair": f"{source_language}|{target_language}",
    }

    try:
        response = httpx.get(
            TRANSLATION_API_URL,
            params=params,
            timeout=TRANSLATION_TIMEOUT_SECONDS
        )
    except httpx.HTTPError:
        return TRANSLATION_NOT_AVAILABLE

    if response.status_code != 200:
        return TRANSLATION_NOT_AVAILABLE

    try:
        data = response.json()
    except ValueError:
        return TRANSLATION_NOT_AVAILABLE

    translated_text = data.get("responseData",
{}).get("translatedText")

    if not isinstance(translated_text, str) or not
translated_text.strip():
        return TRANSLATION_NOT_AVAILABLE

    translated_text = translated_text.strip()

    if translated_text.lower() in {"", "null", "none", "n/a"}:
        return TRANSLATION_NOT_AVAILABLE

    return translated_text
```

Лістинг Б.16 – Приклад конфігурації змінних середовища (.env)

```
DATABASE_URL=mysql+pymysql://user:password@localhost:3306/reading_
app
JWT_SECRET_KEY=replace-with-a-long-random-secret
JWT_ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=60
```