КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет автоматизації інформаційних технологій

Кафедра інформаційних технологій

(назва випускової кафедри)

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

на тему:

Розробка веб додатку бібліотечного контенту на основі онтологічного підходу

Частина перша: Розробка API та бази данних

ЗУБ Микита Романович

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ
Факультет автоматизації інформаційних технологій
Інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
Тетяна Гончаренко
_____
„___”_____2025 року

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА ЗДОБУТТЯ
ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

Розробка веб додатку бібліотечного контенту на основі онтологічного підходу
Частина перша: Розробка API та бази данних

Виконав
Зуб Микита Романович
122 Комп’ютерні науки
(спеціальність)
Інформаційні управляючі системи та технології
(освітня програма)
Групи КН-21-3
Керівник к.т.н. доц., Горда О.В.
(прізвище та ініціали)
Ідентичність підтверджую

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет:	Автоматизації інформаційних технологій
Випускова кафедра:	Інформаційних технологій
Освітній ступінь:	Бакалавр
Спеціальність:	Комп'ютерні науки
Освітня програма:	Інформаційні управляючі системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТ

Тетяна ГОНЧАРЕНКО

_____

„___"_____2025 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА

ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

Зуб Микита Романович

1. Тема роботи Розробка веб додатку бібліотечного контенту на основі онтологічного підходу. Частина перша: Розробка API та бази данних

затверджена наказом ректора КНУБА №235/23/25 від «14» лютого 2025 року

2. Керівник роботи ктн, доц., Горда Олена Володимирівна

3. Строк подання Здобувачем роботи до захисту_____

4. Зміст пояснювальної записки за розділами:

Р.1	АНАЛІЗ ТА ДОСЛІДЖЕННЯ ПРОБЛЕМИ
Р.2	ПРОЕКТУВАННЯ СИСТЕМИ ЕЛЕКТРОННОЇ БІБЛІОТЕКИ
Р.3	РОЗРОБКА СИСТЕМИ ЕЛЕКТРОННОЇ БІБЛІОТЕКИ
Р.4	ЕРГОНОМІЧНІ ПОКАЗНИКИ СИСТЕМИ ЕЛЕКТРОННОЇ БІБЛІОТЕКИ

5. Графічний матеріал за розділами:

1. Актуальність, обєкт, предмет, методи дослідження. 2. Дерево цілей. 3. Аналіз існуючих розробок і вимоги. 4. Модель "чорна скринька". 5. Структура системи. 6. Структура бази данних. 7. Модель управління контентом. 8. Стек технологій та загальна архітектура API. 9. Описочових модулів. 10. Ергономіка системи. 11. Контрольний приклад. 12. Висновки

6. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1	04.02.2025
Розділ 2	15.03.2025
Розділ 3	18.04.2025
Розділ 4	1.05.2025
Остаточне оформлення роботи	17.05.2025
Направлення роботи для перевірки на плагіат	26.05.2025
Попередній захист роботи на випусковій кафедрі	
Направлення роботи на рецензування	

7. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта	Перевірив	
		дата	підпис
Розділ 1	Горда О.В.		
Розділ 2	Горда О.В.		
Розділ 3	Горда О.В.		
Розділ 4	Рябчун Ю.В.		

8. Дата видачі завдання 3 грудня 2024

Зав. кафедри Гончаренко Т.А.

(підпис) (прізвище та ініціали)

Керівники Горда О.В.

(підпис) (прізвище та ініціали)

Здобувач Зуб М.Р.

(підпис) (прізвище та ініціали)

<table>
<tr><td>РЕЗЮМЕ (SUMMARY)
до атестаційної
випускної роботи
Здобувача:</td><td colspan="3">Зуб Микита Романович
Mykyta Zub</td></tr>
<tr><td>ЗВО</td><td colspan="3">Київський національний університет будівництва і архітектури</td></tr>
<tr><td>Тема (українською та англійською)</td><td colspan="3">Розробка веб додатку бібліотечного контенту на основі онтологічного підходу.
Частина перша: Розробка API та бази данних
Developing a web application for library content based on an ontological approach.
Part one: API and database development</td></tr>
<tr><td>Освітній ступінь</td><td colspan="3">Бакалавр</td></tr>
<tr><td>Факультет</td><td colspan="3">Автоматизації і інформаційних технологій</td></tr>
<tr><td>Випускаюча кафедра</td><td colspan="3">Інформаційних технологій</td></tr>
<tr><td>Спеціальність</td><td colspan="3">122 «Комп’ютерні науки»</td></tr>
<tr><td>Освітня програма</td><td colspan="3">Інформаційні управляючі системи та технології</td></tr>
<tr><td>Керівник</td><td colspan="3">Горда Олена Володимирівна</td></tr>
<tr><td rowspan="2">Обсяг роботи:</td><td>пояснювальна записка, стор.</td><td>розділів</td><td>кількість рисунків</td></tr>
<tr><td>165</td><td>4</td><td>23</td></tr>
<tr><td>Ключові слова:

Keywords:</td><td colspan="3">веб-додаток, бібліотечний контент, онтологічний підхід, семантичне моделювання, розробка API, база даних, RESTful сервіс, структура даних, OWL-онтологія, RDF-представлення, JSON-обмін

web application, library content, ontological approach, semantic modeling, API development, database, RESTful service, data structuring, OWL ontology, RDF representation, JSON exchange</td></tr>
</table>

У роботі проведено класифікацію типів бібліотечного контенту та проаналізовано вимоги до їх семантичного подання на основі онтологічного підходу. З метою створення веб додатку бібліотечного контенту для цифровізації бібліотечного середовища, розроблено програмний продукт, що автоматизує процес моделювання контенту через RESTful API та реляційну базу даних.

Здобувач: _____ /Микита ЗУБ/

Керівник: _____/ Олена ГОРДА /

“___” _____ 2025 р.

ЗМІСТ6

ВСТУП

Актуальність дослідження. У сучасному інформаційному суспільстві обсяг даних зростає з надзвичайною швидкістю, що створює численні виклики для їх ефективного зберігання, організації та доступу. Цей феномен обумовлений не лише збільшенням кількості інформаційних джерел, але й різноманітністю форматів контенту, які потребують специфічних підходів до їх обробки. Бібліотеки, як традиційні центри знань і культурної спадщини, стикаються з необхідністю адаптації до цих змін, щоб залишатися актуальними та корисними для користувачів різних категорій. Зокрема, сучасні користувачі очікують миттєвого доступу до інформації, зручних інструментів для її пошуку та можливості інтеграції різних типів контенту в одному середовищі.

У зв'язку з цим, застосування онтологічного підходу в управлінні бібліотечним контентом є перспективним рішенням, яке дозволяє створювати гнучкі моделі даних, адаптовані до змінних вимог користувачів. Онтології не лише структурують інформацію, але й забезпечують її семантичне збагачення, що покращує пошук та навігацію, роблячи систему більш інтуїтивно зрозумілою для користувачів. Завдяки онтологіям, бібліотеки можуть не просто зберігати інформацію, а й надавати користувачам можливість розуміння контексту та взаємозв'язків між різними інформаційними об'єктами, що є критично важливим в умовах інформаційного перевантаження.

Мета дослідження. Розробка web-додатку для управління бібліотечним контентом на основі онтологічного підходу. Цей додаток має забезпечити інтеграцію різноманітних типів контенту, включаючи книги, мультимедійні ресурси та наукові дані, з метою покращення їх доступності та взаємодії з користувачами. Використання онтологій сприятиме визначенню взаємозв'язків між елементами контенту, що дозволить користувачам легше знаходити та використовувати релевантну інформацію. Крім того, такий підхід забезпечить

можливість адаптації системи до нових типів даних у майбутньому, що є важливим для підтримки її актуальності.

Об'єкт дослідження. Web-додаток для управління бібліотечним контентом, який використовує онтології для організації та семантичного збагачення даних. Це забезпечує ефективну інтеграцію та доступ до різноманітних інформаційних ресурсів, що є ключовим для задоволення потреб сучасних користувачів. Особливу увагу приділено тому, як онтології можуть бути використані для поліпшення існуючих процесів каталогізації та пошуку інформації.

Предмет дослідження. Методології, методи та інструментальні засоби розробки та впровадження онтологічно-орієнтованого web-додатку, включаючи побудову архітектури системи, розробку функціональних компонентів та інтеграцію з існуючими бібліотечними системами. Дослідження охоплює аналіз технічних і організаційних аспектів, які впливають на успішність впровадження таких систем, а також оцінку їхньої ефективності в реальних умовах експлуатації.

Методи дослідження. Для досягнення мети та виконання завдань дослідження були застосовані такі методи:

- Аналіз літературних джерел та інформаційних ресурсів. Проведено детальне вивчення наукових праць, монографій, статей та технічної документації, присвячених тематиці онтологій, семантичних технологій та управління бібліотечними системами. Це дозволило визначити сучасні тенденції, основні виклики та перспективи використання онтологічного підходу для організації бібліотечного контенту. Особливу увагу приділено аналізу міжнародних стандартів у сфері метаданих та практичних прикладів впровадження подібних рішень.
- Метод структурного аналізу та моделювання. Цей метод використовувався для створення концептуальної моделі, яка відображає структуру та взаємозв'язки між елементами бібліотечного контенту. На основі аналізу вимог користувачів і технічних можливостей було побудовано модель, яка забезпечує інтеграцію різних типів даних,

включаючи текстові, графічні та мультимедійні ресурси. Моделювання також охоплювало розробку онтологічної структури, яка сприяє семантичному збагаченню інформації та покращенню процесів пошуку й навігації.

- Метод системного проектування. Використано для розробки загальної архітектури web-додатку, його функціональних компонентів і основних модулів. На основі концептуальної моделі було спроектовано структуру системи, яка включає модулі для введення, зберігання, пошуку та візуалізації даних. Значна увага приділялася адаптивності системи до різних типів даних і її інтеграції з існуючими бібліотечними платформами. Процес проектування включав також підготовку до впровадження онтологій у практичну діяльність бібліотек.

Ці методи забезпечили комплексний підхід до виконання поставлених завдань і дозволили реалізувати науково обґрунтовані рішення для створення інноваційного web-додатку.

Практична значимість. Результати дослідження можуть бути використані для створення ефективних систем управління бібліотечним контентом, що відповідають сучасним вимогам до організації інформації. Такий підхід сприятиме підвищенню якості обслуговування користувачів бібліотек, забезпечуючи їм доступ до релевантного та семантично збагаченого контенту. Крім того, впровадження онтологічного підходу може стати основою для розвитку нових послуг та функціональностей, які задовольнять зростаючі потреби користувачів у цифрову епоху. Це включає в себе можливість персоналізації користувацького досвіду, підтримку багатомовності та інтеграцію з глобальними інформаційними мережами.

1. АНАЛІЗ ТА ДОСЛІДЖЕННЯ ПРОБЛЕМИ

1.1. Аналіз проблеми

У сучасному інформаційному суспільстві бібліотеки відіграють ключову роль як центри знань та інформації. Проте, з розвитком цифрових технологій, вони стикаються з численними викликами, пов'язаними з ефективним управлінням та доступом до великого обсягу різноманітного контенту. Традиційні системи каталогізації часто не можуть забезпечити гнучкість та адаптивність, необхідні для задоволення зростаючих потреб користувачів. Це призводить до ускладнень у пошуку та навігації, що негативно впливає на користувацький досвід.

- Проблеми традиційних бібліотечних систем

Традиційні бібліотечні системи зазвичай базуються на жорстких структурах даних, які не враховують семантичні зв’язки між різними типами контенту. Це ускладнює інтеграцію нових ресурсів, таких як електронні книги, аудіо- та відеоматеріали, а також знижує ефективність пошуку. Крім того, відсутність стандартизованих підходів до опису та організації даних може призвести до дублювання інформації та зниження її якості. Це особливо актуально в умовах, коли обсяги інформації постійно зростають, і бібліотеки повинні забезпечувати швидкий та точний доступ до різноманітних ресурсів.

- Виклики інтеграції нових технологій

З появою нових технологій, таких як великі дані, машинне навчання та штучний інтелект, бібліотеки мають можливість покращити свої системи управління контентом. Однак, інтеграція цих технологій вимагає значних зусиль, оскільки потребує адаптації існуючих систем та процесів. Бібліотеки повинні розробляти нові стратегії для впровадження технологій, які дозволять їм ефективно управляти великими обсягами даних та забезпечувати більш інтуїтивний доступ до інформації.

- Онтологічний підхід як рішення

Онтологічний підхід пропонує вирішення цих проблем шляхом створення структурованих моделей, які відображають взаємозв’язки між елементами контенту. Онтології дозволяють формалізувати знання в певній галузі, забезпечуючи можливість більш інтуїтивного та точного пошуку, а також полегшуючи інтеграцію нових даних. Це особливо важливо для бібліотек, які прагнуть забезпечити користувачам доступ до різноманітних ресурсів, враховуючи їхні семантичні зв’язки.

- Актуальність проблеми

У сучасному світі обсяги інформації зростають експоненціально, що створює значні виклики для бібліотек у забезпеченні ефективного доступу до знань. Традиційні методи організації та управління інформацією стають менш ефективними, що може призвести до втрати цінних даних та зниження якості обслуговування користувачів. Впровадження онтологічного підходу стає необхідністю для модернізації бібліотечних систем, оскільки він дозволяє створювати більш гнучкі та адаптивні структури даних, що відповідають сучасним вимогам інформаційного суспільства. Застосування онтологій у бібліотеках сприяє покращенню пошуку та навігації, підвищенню якості інформаційних послуг та забезпеченню інтеграції різноманітних ресурсів, що є критично важливим у контексті швидкого розвитку цифрових технологій.

- Етапи впровадження онтологічного підходу

Впровадження онтологічного підходу може бути розділене на кілька ключових етапів:

1. Аналіз та планування: На цьому етапі важливо визначити цілі та завдання системи, а також провести аналіз існуючих технологій та методів, які можуть бути використані для вирішення проблеми.
2. Збір та аналіз вимог: Визначення вимог користувачів та специфікацій системи є критично важливим для успішного впровадження онтологічного підходу. Це включає в себе вивчення потреб користувачів, аналіз існуючих процесів та визначення ключових функціональних можливостей системи.

3. Проектування онтологічної моделі: Розробка архітектури та структури онтологічної моделі є важливим кроком у напрямку забезпечення семантичного збагачення даних. Це включає в себе визначення ключових концепцій, відношень та атрибутів, які будуть використовуватися для опису контенту.
4. Розробка та інтеграція: На цьому етапі відбувається створення програмного забезпечення, яке реалізує онтологічний підхід, а також його інтеграція з існуючими системами. Це може включати в себе розробку нових інтерфейсів, модулів та сервісів, які забезпечать ефективне управління контентом.
5. Тестування та валідація: Перевірка функціональності та ефективності системи є важливим етапом, який дозволяє виявити та усунути можливі помилки та недоліки. Це включає в себе тестування системи на відповідність вимогам, перевірку її продуктивності та безпеки.
6. Впровадження та обслуговування: Після успішного тестування система може бути впроваджена в експлуатацію. На цьому етапі важливо забезпечити належне обслуговування системи, включаючи моніторинг її роботи, оновлення та підтримку користувачів.

- Перспективи розвитку

Незважаючи на виклики, пов'язані з впровадженням онтологічного підходу, його реалізація може значно покращити користувацький досвід, підвищити ефективність пошуку та забезпечити більш точну інтеграцію нових ресурсів. У майбутньому, розвиток таких технологій може стати основою для створення інноваційних бібліотечних систем, які будуть відповідати зростаючим вимогам інформаційного суспільства.

Таким чином, проблема ефективного управління бібліотечним контентом залишається актуальною, і вирішення її за допомогою онтологічного підходу може стати важливим кроком у напрямку покращення якості бібліотечних послуг. Це відкриває нові можливості для бібліотек у забезпеченні доступу до знань та інформації, що є ключовим фактором у розвитку сучасного суспільства.

1.2. Визначення цілей дослідження

Основною метою роботи є отримання функціональної серверної частини web-додатку для управління бібліотечним контентом з використанням онтологічного підходу. Дерево цілей опису даної мети наведено на рисунку 1.1.

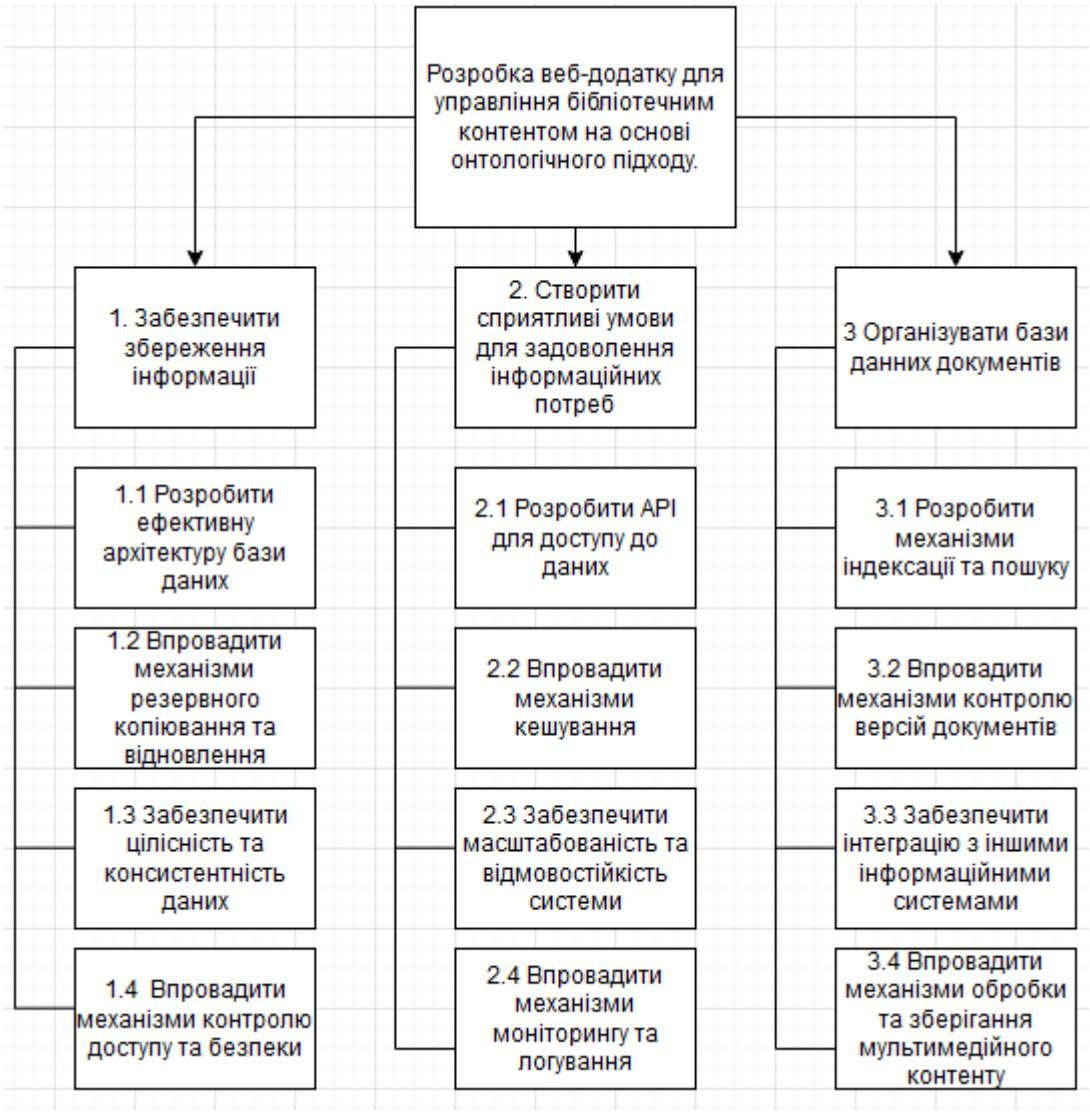


Рис. 1.1. Дерево цілей

З поставленої мети випливають задачі що наведені на рисунку 1.2

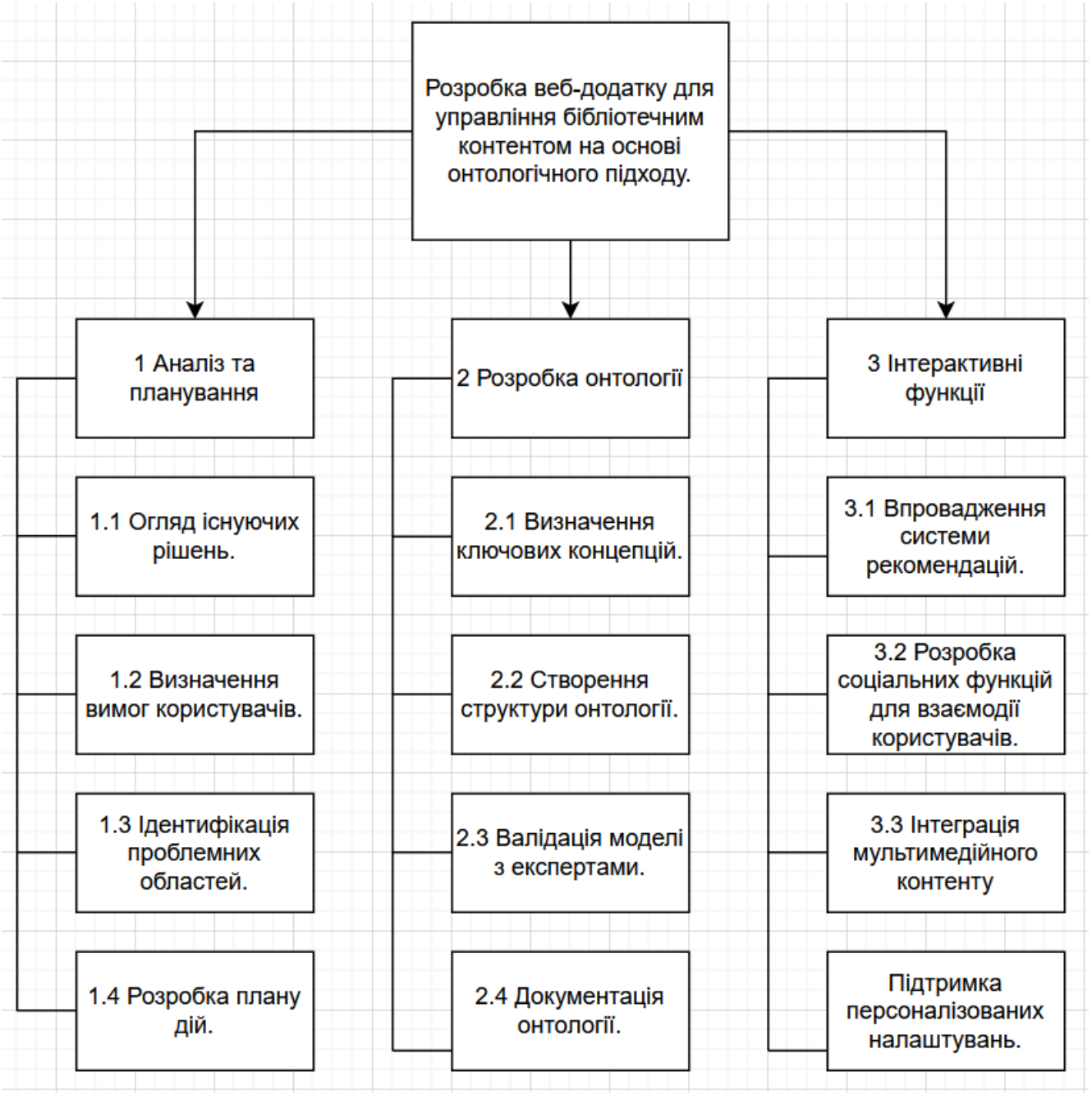


Рис. 1.2. Дерево задач

1.3. Аналіз існуючих розробок автоматизованих систем бібліотек з використанням онтологій та систем нечіткого пошуку

1.3.1. Історія розвитку цифрових бібліотек

Цифрові бібліотеки пройшли значний шлях розвитку від перших ініціатив до сучасних глобальних проєктів, що забезпечують доступ до знань мільйонам користувачів по всьому світу.

Одним із перших проєктів у сфері електронних бібліотек став Проєкт «Гутенберг», започаткований у 1971 році. Його метою було створення електронних версій літературних творів, доступних для вільного використання. У 1994 році в Росії з'явилася «Бібліотека Максима Мошкова», яка стала однією з найбільших електронних бібліотек російськомовного сегменту Інтернету.

У 1990 році Бібліотека Конгресу США розпочала проєкт «Пам'ять Америки», який надавав вільний доступ до електронних матеріалів з історії США. У Європі в 1995 році під егідою Ради Європи було започатковано проєкт «Bibliotheca Universalis», метою якого було створення глобальної мережі електронних бібліотек.

У 2002 році компанія Google розпочала власний проєкт з оцифрування книг, який у 2004 році переріс у бібліотечний проєкт Google Print, а згодом був перейменований на Пошук книг Google. У 2008 році розпочала функціонувати загальноєвропейська цифрова бібліотека Europeana, яка на момент відкриття містила 2 мільйони оцифрованих об'єктів.

У 2009 році відбулося офіційне відкриття Світової цифрової бібліотеки, створеної за підтримки ЮНЕСКО та Бібліотеки Конгресу США, яка надає вільний доступ до культурних скарбів з усього світу.

В Україні процес формування електронних бібліотек активізувався у 1990-х роках. Національна бібліотека України імені В. І. Вернадського стала піонером у цій сфері, започаткувавши оцифрування своїх фондів та надаючи доступ до електронних каталогів. Законодавча база, зокрема Закон України «Про національну програму інформатизації» (1998 рік), сприяла розвитку електронних ресурсів у країні.

Сьогодні цифрові бібліотеки є невід'ємною частиною інформаційного суспільства. Вони забезпечують швидкий та зручний доступ до величезних масивів інформації, сприяють збереженню культурної спадщини та підтримують наукові дослідження. Завдяки розвитку інформаційно-комунікаційних технологій,

бібліотеки інтегруються в електронне середовище, надаючи користувачам нові можливості для навчання та саморозвитку.

Таким чином, цифрові бібліотеки пройшли еволюційний шлях від перших експериментальних проєктів до потужних інформаційних ресурсів, що об'єднують мільйони документів та забезпечують глобальний доступ до знань.

1.3.2. Аналіз теоретичних досліджень в галузі цифрових бібліотек

Аналіз теоретичних досліджень у галузі цифрових бібліотек свідчить про їхній стрімкий розвиток, починаючи з 1990-х років. У цей період західні бібліотеки, музеї та архіви розпочали створення цифрових колекцій шляхом сканування паперових документів. Заснування у 1995 році мережевого видання «D-Lib Magazine» стало важливим кроком у вивченні та розвитку електронних бібліотек, охоплюючи перспективні інформаційні технології, програмні засоби та соціально-економічні аспекти створення цифрового контенту.

Варто зазначити, що спеціалізовані науково-дослідні центри з вивчення електронних бібліотек функціонують як у країнах Західної Європи, так і в США. Прикладом є Centre for Digital Library Research (CDLR) в Університеті Стратклайда, Глазго, Велика Британія. У 1997 році в Німеччині розпочав видаватися «International Journal on Digital Libraries», присвячений теорії та практиці комплектування, організації та управління цифровим контентом. Цей журнал публікує матеріали щодо створення електронної інформації, використання високошвидкісних мереж для її передачі, питань інформаційної безпеки та користувацьких інтерфейсів.

У 2003 році з'явився щоквартальний журнал «Journal of Digital Information Management», який, окрім загальних питань формування та функціонування цифрового інформаційного середовища, включає статті з управління цифровою інформацією, архівування даних та електронних бібліотек. У концептуальній статті А. Ширі (2003) були окреслені основні напрями вивчення електронних бібліотек.

Сучасні дослідження в цій галузі зосереджені на управлінні знаннями, що базується на використанні тезаурусів та класифікаційних систем для перехресного перегляду та пошуку в різних цифрових колекціях; створенні онтологій для представлення знань; застосуванні таксономій для забезпечення уніфікованого доступу до контенту різних цифрових депозитаріїв. Особливу увагу приділяють формуванню «бібліотечного простору», який інтегрує фізичні та цифрові ресурси для покращення доступу до інформації.

Дослідження поведінки користувачів та їхніх потреб у цифровій інформації в різних контекстах, включаючи університетське середовище, школи, урядові установи та бізнес, є важливим напрямом для покращення якості системного проєктування електронних бібліотек. Вивчаються аспекти простоти використання, доступності та задоволеності користувачів конкретними електронними бібліотеками; підтримка навчання, викладання та дослідницької діяльності через інтеграцію віртуальних навчальних середовищ та цифрових бібліотек; оцінка поведінки різних користувацьких спільнот на основі їхніх знань, вікових характеристик та специфічних потреб.

Однією з основних проблем у дослідженнях користувачів електронних бібліотек є методика та прийоми збору даних. Виявлено, що користувачі електронних бібліотек застосовують традиційні методи читацької аудиторії, що вимагає адаптації дослідницьких підходів до специфіки цифрового середовища.

Загалом, проблематика електронних бібліотек має комплексний характер, оскільки цей феномен розглядається в межах різних галузей знань, включаючи інформаційні технології, бібліотекознавство, соціологію та інші дисципліни. Подальші дослідження спрямовані на вдосконалення технологій управління цифровим контентом, покращення користувацького досвіду та інтеграцію електронних бібліотек у глобальне інформаційне суспільство.

1.3.3 Аналіз існуючих розробок в галузі цифрових бібліотек

У сучасному інформаційному суспільстві автоматизовані бібліотечні інформаційні системи (АБІС) відіграють ключову роль у забезпеченні ефективного доступу до знань та інформаційних ресурсів. Вони дозволяють бібліотекам автоматизувати процеси комплектування, каталогізації, обслуговування користувачів та управління фондами.

У теоретичному аспекті автоматизовані бібліотечні інформаційні системи (АБІС) розвиваються шляхом інтеграції сучасних інформаційних технологій, таких як онтології та нечіткий пошук, для підвищення ефективності управління знаннями та покращення пошукових можливостей.

Використання онтологій: Онтології служать для формалізації знань у певній предметній області, що дозволяє створювати семантично багаті моделі даних. У контексті АБІС це сприяє покращенню каталогізації, індексації та пошуку інформації, забезпечуючи більш точне та релевантне відображення взаємозв'язків між різними інформаційними ресурсами. Застосування онтологій у бібліотечних системах також полегшує інтеграцію з іншими інформаційними системами та підтримує розвиток семантичного web-у.

Системи нечіткого пошуку: Нечіткий пошук базується на принципах нечіткої логіки, що дозволяє обробляти неточні або неповні запити користувачів. У бібліотечних системах це особливо корисно для пошуку документів за неповними або приблизними даними, враховуючи можливі орфографічні помилки або варіації у формулюванні запитів. Використання нечіткого пошуку підвищує зручність та ефективність взаємодії користувачів з бібліотечними ресурсами.

Інтеграція онтологій та нечіткого пошуку в АБІС: Поєднання онтологічного підходу з методами нечіткого пошуку дозволяє створювати більш адаптивні та інтелектуальні системи, здатні враховувати контекст та індивідуальні потреби користувачів. Це сприяє розвитку персоналізованих сервісів у бібліотеках, підвищуючи загальну якість обслуговування та доступу до інформації.

Таким чином, теоретичні дослідження в галузі автоматизованих бібліотечних систем зосереджуються на впровадженні онтологій та нечіткого пошуку для покращення управління знаннями, підвищення точності та релевантності пошукових результатів, а також забезпечення більш гнучкої та інтуїтивно зрозумілої взаємодії користувачів з інформаційними ресурсами бібліотек**.**

1. Koha — це перша відкрита автоматизована бібліотечна система, розроблена у 1999 році. Вона підтримує всі основні бібліотечні процеси, включаючи каталогізацію, обслуговування користувачів, управління фондами та звітність. Koha відома своєю гнучкістю та можливістю адаптації до потреб різних бібліотек.

2. Aleph — це комерційна бібліотечна система, розроблена компанією Ex Libris. Вона забезпечує інтегроване управління бібліотечними ресурсами, підтримуючи різноманітні формати даних та стандарти каталогізації. Aleph широко використовується в академічних та наукових бібліотеках по всьому світу.

3. Evergreen — це відкрита автоматизована бібліотечна система, розроблена для консорціумів та великих бібліотечних систем. Вона забезпечує масштабованість та надійність, підтримуючи функції каталогізації, обслуговування користувачів, управління фондами та звітності.

4. Sierra — це комерційна бібліотечна система, розроблена компанією Innovative Interfaces. Вона поєднує в собі традиційні бібліотечні функції з сучасними технологіями, забезпечуючи інтеграцію з електронними ресурсами та підтримку мобільних пристроїв.

5. Alma — це хмарна бібліотечна платформа, розроблена компанією Ex Libris. Вона об'єднує управління друкованими, електронними та цифровими ресурсами, забезпечуючи єдиний інтерфейс для всіх бібліотечних операцій. Alma підтримує сучасні стандарти та протоколи, що робить її привабливою для великих академічних бібліотек.

6. Horizon — це комерційна бібліотечна система, розроблена компанією SirsiDynix. Вона забезпечує управління бібліотечними ресурсами, включаючи

каталогізацію, обслуговування користувачів та управління фондами. Horizon відома своєю надійністю та гнучкістю в налаштуванні.

Порівняльні характеристики бібліотек наведені у таблиці 1.1.

Таблиця 1.1.

Порівняння бібліотечних систем

Назва	Тип ліцензії	Основні функції	Цільова аудиторія
Koha	Відкрита	Каталогізація, обслуговування користувачів, управління фондами, звітність	Публічні та академічні бібліотеки
Aleph	Комерційна	Інтегроване управління ресурсами, підтримка різних форматів та стандартів	Академічні та наукові бібліотеки
Evergreen	Відкрита	Масштабованість, каталогізація, обслуговування користувачів, управління фондами	Консорціуми та великі бібліотеки
Sierra	Комерційна	Інтеграція з електронними ресурсами, підтримка мобільних пристроїв	Публічні та академічні бібліотеки
Alma	Комерційна	Управління друкованими, електронними та цифровими ресурсами, хмарна платформа	Великі академічні бібліотеки
Horizon	Комерційна	Каталогізація, обслуговування користувачів, управління фондами	Публічні та спеціалізовані бібліотеки

Ці системи відрізняються за типом ліцензії, функціональністю та цільовою аудиторією, що дозволяє бібліотекам обирати найбільш підходяще рішення відповідно до своїх потреб та ресурсів.

1.4. Аналіз вимог та особливостей “Електронна бібліотека”

Закон України «Про бібліотеки і бібліотечну справу» визначає правові та організаційні засади функціонування бібліотек в Україні. Він встановлює статус бібліотек, їхні основні завдання, права та обов'язки, а також регулює питання формування та збереження бібліотечних фондів. Закон гарантує право громадян на вільний доступ до інформації та знань, що зберігаються в бібліотеках, сприяючи задоволенню інформаційних, наукових і культурних потреб суспільства.

Положення про Українську цифрову бібліотеку визначає основні засади створення та функціонування цієї інформаційної системи. Українська цифрова бібліотека призначена для накопичення, упорядкування, обліку, обробки, зберігання та використання фондів електронних документів, а також для обслуговування користувачів через телекомунікаційні мережі. Положення регламентує структуру, функції та порядок доступу до ресурсів бібліотеки, забезпечуючи дотримання вимог щодо захисту інтелектуальних та майнових прав.

Закон України «Про основні засади забезпечення кібербезпеки України» встановлює правові та організаційні основи захисту національних інтересів у сфері кібербезпеки. Він визначає об'єкти кіберзахисту, суб'єкти забезпечення кібербезпеки та їхні повноваження, а також основні напрями державної політики у цій сфері. Закон спрямований на створення умов для безпечного функціонування кіберпростору, запобігання кіберзагрозам та реагування на кіберінциденти, що є особливо важливим для захисту інформаційних ресурсів, зокрема електронних бібліотек.

Вимоги до електронної бібліотеки включають забезпечення функціональності, безпеки, стабільності, ефективності та масштабованості системи. Функціональність передбачає реалізацію пошуку ресурсів на основі онтологічних структур із застосуванням алгоритмів нечіткого пошуку, а також навігацію за категоріями та фільтрацію результатів. Безпека охоплює шифрування даних, аутентифікацію та авторизацію користувачів. Стабільність вимагає

підтримки роботи при значних обсягах даних та відмовостійкості системи. Ефективність досягається оптимізацією запитів та використанням сучасних баз даних. Масштабованість забезпечує можливість розширення системи відповідно до зростаючих потреб користувачів та обсягів даних.

Основні вимоги:

1. Функціональність:
    - Реалізація пошуку бібліотечних ресурсів на основі онтологічних структур із застосуванням алгоритмів нечіткого пошуку для обробки неоднозначних або неповних запитів.
    - Забезпечення навігації за категоріями та фільтрації результатів на основі онтологічних зв’язків.
2. Безпека:
    - Гарантування конфіденційності даних користувачів через шифрування запитів та результатів пошуку (наприклад, використання TLS).
    - Впровадження механізмів аутентифікації та авторизації для контролю доступу до персоналізованих функцій.
3. Стабільність:
    - Підтримка стабільної роботи системи навіть при значних обсягах даних.
    - Забезпечення відмовостійкості через автоматичне відновлення з’єднання та кешування результатів пошуку.
4. Ефективність:
    - Оптимізація запитів за допомогою індексів онтологічної бази даних.
    - Використання алгоритмів нечіткого пошуку, таких як методи порівняння за схожістю слів (наприклад, алгоритм Левенштейна).

Особливості проектування:

1. Приватність:
    - Застосування шифрування запитів та результатів пошуку (AES, RSA).

- Реалізація аутентифікації через паролі або двофакторну аутентифікацію.

2. Стабільність:
   - Використання надійних серверних технологій, таких як FAST API
   - Впровадження балансування навантаження та резервування для забезпечення безперебійної роботи.
3. Ефективність:
   - Застосування онтологічних структур для індексації даних.
   - Використання сучасних баз даних та пошукових систем (наприклад, Elasticsearch) для швидкої обробки запитів.
4. Масштабованість:
   - Проектування системи з урахуванням можливості горизонтального та вертикального масштабування для обробки зростаючої кількості користувачів та даних.

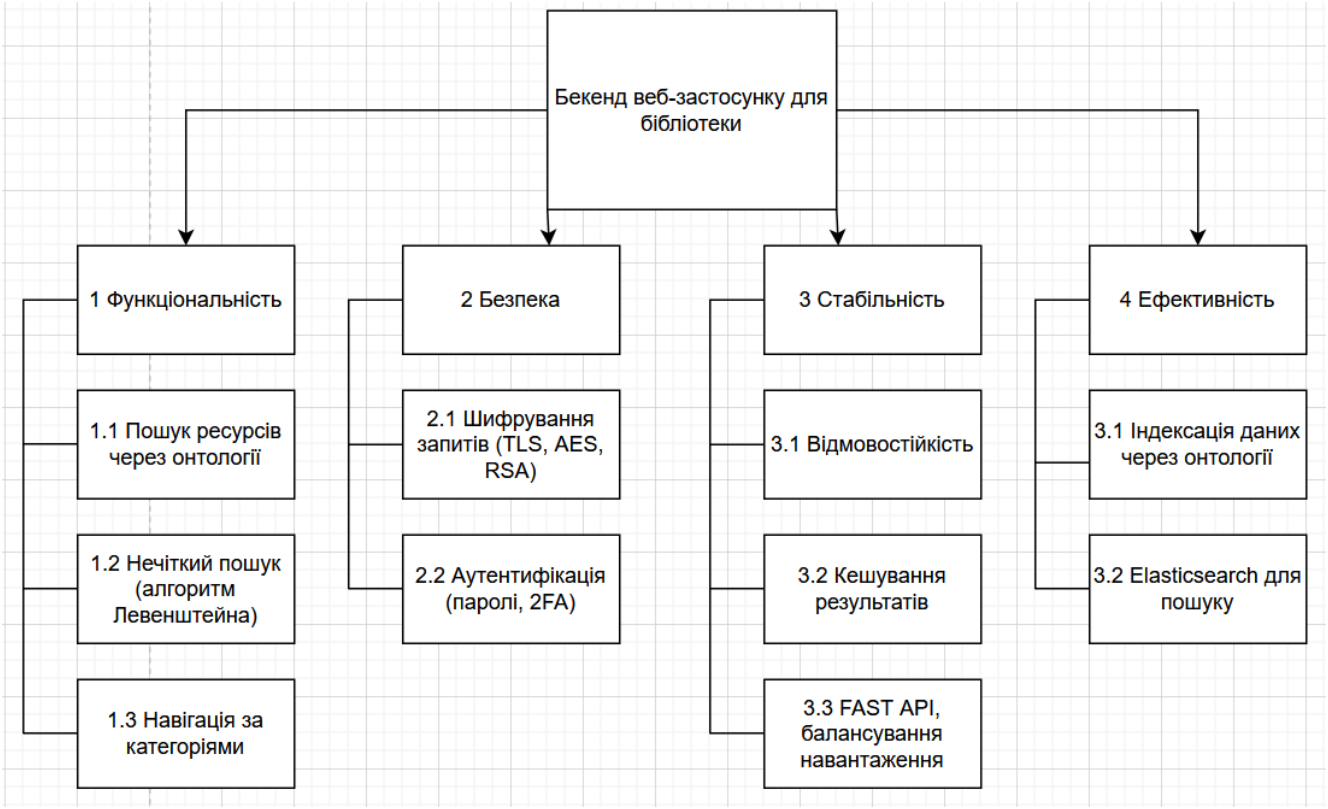


Рис. 1.3. Структура вимог

З урахуванням цих вимог та особливостей, бекенд web-застосунку бібліотеки повинен забезпечувати ефективну обробку запитів, надійне зберігання та пошук інформації, а також гарантувати безпеку та конфіденційність даних користувачів.

- Система**:** Архітектура API та база даних для управління контентом бібліотеки. Визначення типу системи у таблиці 1.2.

- Зовнішнє середовище**:** Клієнтські застосунки, бібліотечні системи, зовнішні API, інформаційні ресурси, інтернет.

Таблиця 1.2.

Тип системи “Електронна бібліотека”

Основа (критерій) класифікації	Класи систем
По взаємодії із зовнішнім середовищем	відкрита
По структурі	складна
По характеру функцій	інтегрована
По характеру розвитку	динамічна
По степені організованості	добре організована
По складності поведінки	адаптивна
По характеру зв’язку між елементами	детермінована
По характеру структури управління	централізована
По призначенню	інформаційна
За походженням	штучна
За типом опису закону функціонування	"білий ящик"

1.5. Аналіз системи “Електронна бібліотека”

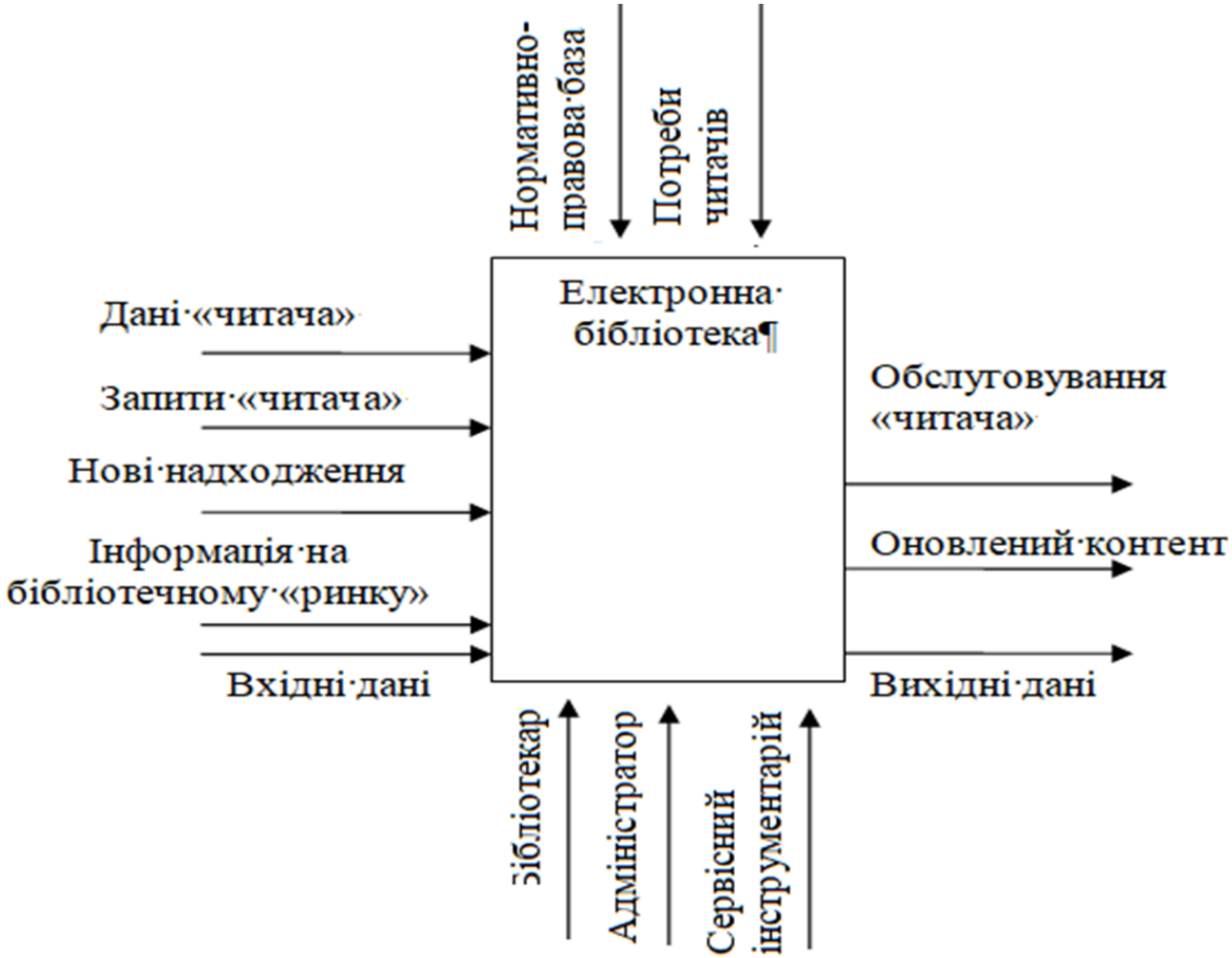


Рис. 1.4. Модель “Чорна скринька”

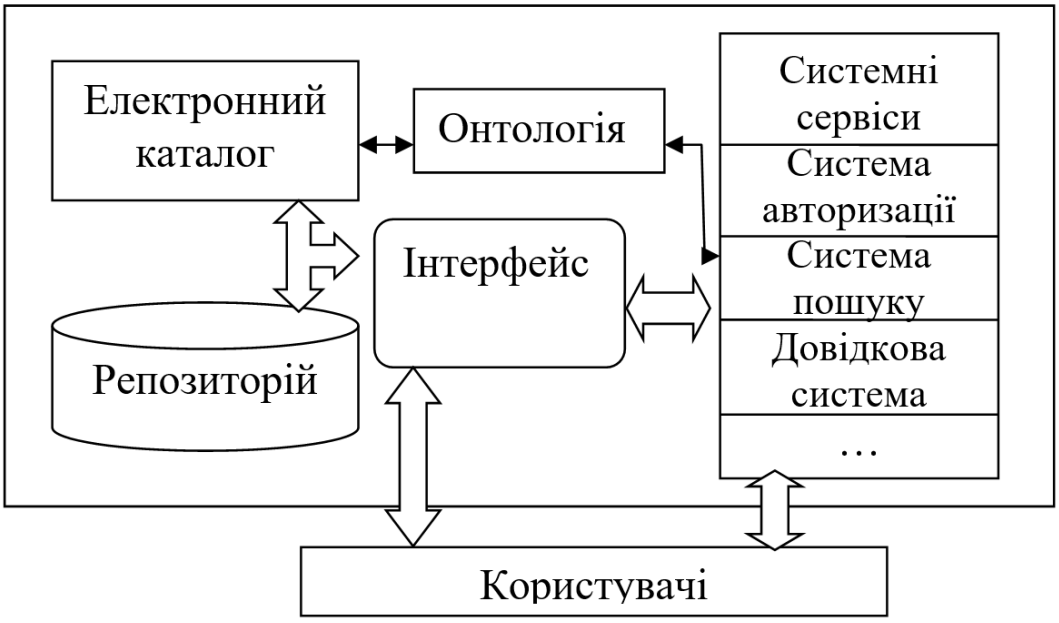


Рис. 1.5. Загальна структура системи

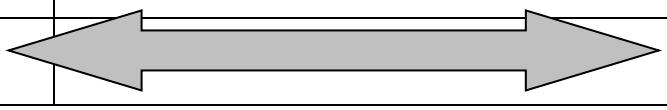
Початковий стан	***Механізм розвитку***	***Вихідний стан***
		
Читачі	Пошукова система	Задоволення потреб читачів
Контент	Інструментарій поповнення та управління контентом	Поповнення контенту
Новини на бібліотечному ринку	Рубрики новин Форуми Вебінари Ведення статистики	Розширення аудитор читачів та ефективності роботи бібліотеки
		

Рис. 1.6. Схема “Стріла цілепокладання”

1.6. Постановка задачі

Завдання полягає в розробці системи управління бібліотечним контентом, яка базується на онтологічному підході, з акцентом на створенні API та бази даних. Основна мета полягає у забезпеченні семантичного збагачення даних, що дозволить підвищити ефективність роботи з бібліотечними ресурсами, забезпечуючи точність та релевантність інформації.

API є центральним компонентом системи, який забезпечує взаємодію між клієнтськими додатками та серверною частиною. Він має бути реалізований у вигляді RESTful сервісу, що підтримує основні операції CRUD (створення, читання, оновлення та видалення даних). Це дозволить користувачам легко додавати нові ресурси, здійснювати пошук інформації, оновлювати існуючі записи та видаляти застарілі дані. Для забезпечення безпеки та конфіденційності інформації необхідно впровадити механізми аутентифікації на основі токенів JWT, які дозволять захистити дані від несанкціонованого доступу. Крім того, важливо забезпечити детальну документацію API, яка спростить інтеграцію та використання для розробників.

Основні функції системи включають кілька ключових процесів. По-перше, це процес додавання контенту, де API забезпечує введення нових даних про бібліотечні ресурси з перевіркою на відповідність онтології. Це гарантує, що дані зберігаються у структурованому та зрозумілому форматі. По-друге, система покращує пошук та навігацію завдяки онтологіям, які враховують семантичні зв’язки між даними, що дозволяє користувачам отримувати більш точні та релевантні результати. По-третє, процес оновлення онтології дозволяє інтегрувати нові концепції та підтримувати актуальність інформації, що є критично важливим для динамічних бібліотечних середовищ. Нарешті, управління доступом включає механізми аутентифікації та авторизації, які контролюють права доступу користувачів до різних частин системи, забезпечуючи безпеку та конфіденційність даних.

Проект стикається з певними викликами, такими як забезпечення семантичної цілісності даних, що вимагає використання спеціалізованих форматів та технологій. Це включає інтеграцію онтологій та забезпечення їхньої актуальності в контексті постійно змінюваних даних. Захист даних також є пріоритетом, тому необхідно впровадити надійні механізми безпеки, такі як SSL/TLS, для захисту даних при передачі. Масштабованість системи повинна бути врахована на етапі проектування, щоб забезпечити можливість обробки зростаючого обсягу даних та запитів без втрати продуктивності.

Таким чином, розробка API та бази даних для web-додатку бібліотечного контенту на основі онтологічного підходу є важливим завданням, яке спрямоване на створення сучасної інформаційної системи. Це рішення має забезпечити семантичне збагачення даних, покращити якість пошуку та управління інформацією, а також гарантувати безпеку та конфіденційність даних. Інтеграція онтологій та використання сучасних технологій web-у дозволять створити ефективну та надійну систему, що відповідає вимогам сучасних бібліотек.

2. ПРОЕКТУВАННЯ СИСТЕМИ ЕЛЕКТРОННОЇ БІБЛІОТЕКИ

2.1. Аналіз процесів системи

Так як система є структурно то функціонально складною, що вимагає розглянути цю систему більш детально з різних сторін за допомогою методик та методів структурного та функціонального аналізу. Першим кроком являється структурний аналіз системи, розглянемо з позиції процесів.

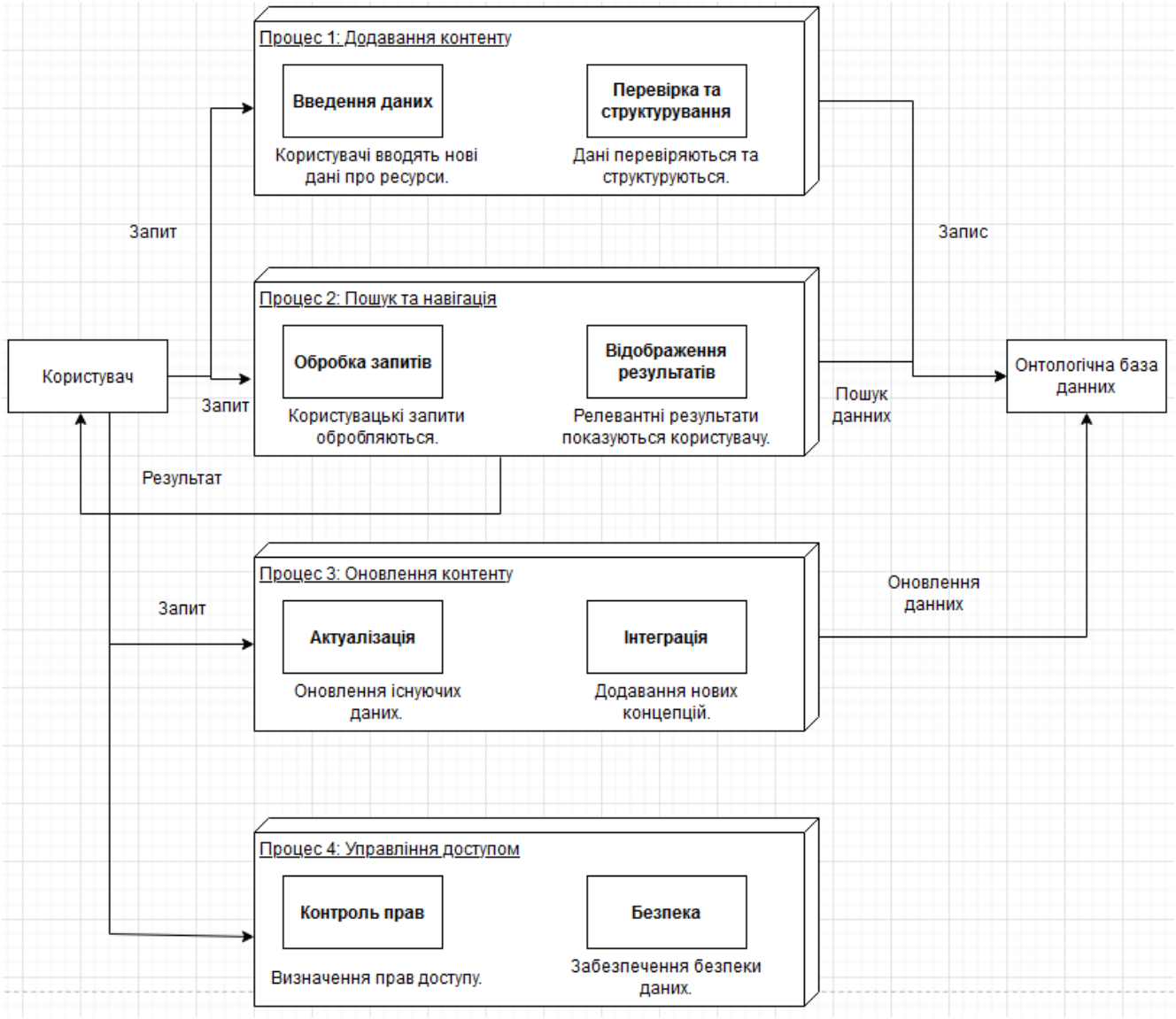


Рис. 2.1. Структура функціональної частини системи

На верхньому рівні діаграми (рис. 2.1.) зображено чотири основні процеси роботи системи веб-додатку для управління бібліотечним контентом.

1. Процес - Додавання контенту:
    - здійснює введення нових даних про бібліотечні ресурси;
    - включає перевірку та структурування даних відповідно до онтології;
    - записує інформацію до онтологічної бази даних.
2. Процес - Пошук та навігація:
    - обробляє запити користувачів на пошук інформації;
    - використовує онтології для покращення точності результатів;
    - відображає релевантні результати користувачу.
3. Процес - Оновлення контенту:
    - відповідає за актуалізацію онтологічної бази даних;
    - інтегрує нові концепції та зв’язки;
    - забезпечує підтримку актуальності даних.
4. Процес - Управління доступом:
    - контролює права доступу користувачів до різних частин системи;
    - забезпечує безпеку та конфіденційність даних;
    - включає механізми аутентифікації та авторизації.

Проведемо більш детальну декомпозицію першого процесу – Додавання контенту(рис. 2.2.).

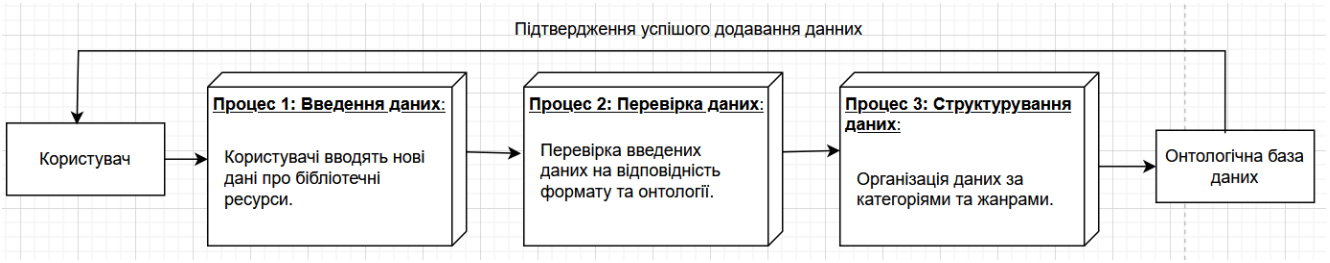


Рис. 2.2. Декомпозиція процесу Додавання контенту.

Маємо наступну послідовність дій на цьому рівні: користувач вводить нові дані про бібліотечні ресурси в спеціально відведене для цього поле. Дані перевіряються на відповідність формату та онтології, після чого вони структуруються за категоріями та зв’язками. Потім ці структуровані дані передаються для збереження в онтологічну базу. Якщо перевірка пройшла успішно,

інформація додається до існуючих записів, забезпечуючи актуальність і повноту бази даних. У разі невідповідності, користувачу надається зворотний зв'язок для корекції введених даних.

Проведемо більш детальну декомпозицію процесу Пошуку та Навігації(рис. 2.3.).

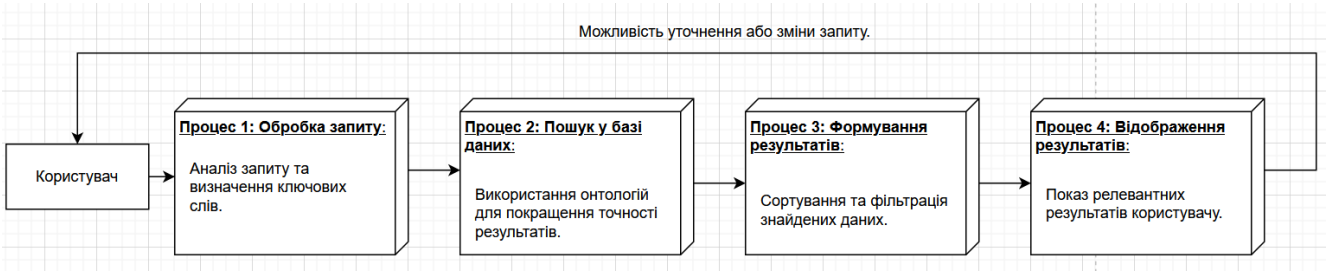


Рис. 2.3. Декомпозиція процесу Пошуку та Навігації.

Відбувається перевірка необхідності пошуку відповідних даних. Відсіюються всі запити, які були сформовані некоректно або не відповідають контексту (наприклад, запити в діловому документі чи внутрішній програмі бібліотеки).

Після прийняття рішення системою, запит може бути відхилений одразу або переданий на подальшу обробку. Результати перевірки передаються системі для виконання наступних дій.

Проведемо більш детальну декомпозицію процесу Оновлення Контенту (рис. 2.4.).

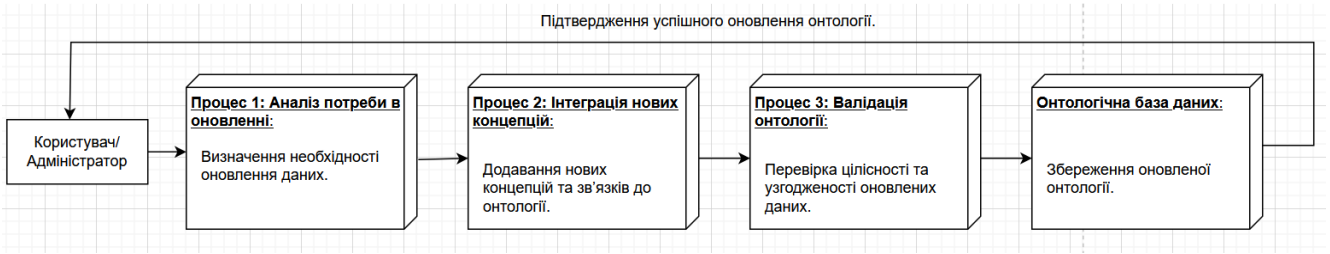


Рис. 2.4. Декомпозиція процесу Оновлення Контенту.

Відбувається перевірка необхідності оновлення онтологічної бази даних. Відсіюються всі дані, які не потребують актуалізації або не відповідають новим концепціям і зв’язкам.

Після прийняття рішення системою, дані можуть бути відхилені одразу або передані на подальшу обробку. Результати перевірки передаються системі для виконання наступних дій.

Проведемо більш детальну декомпозицію процесу Управління Доступом(рис. 2.5.).

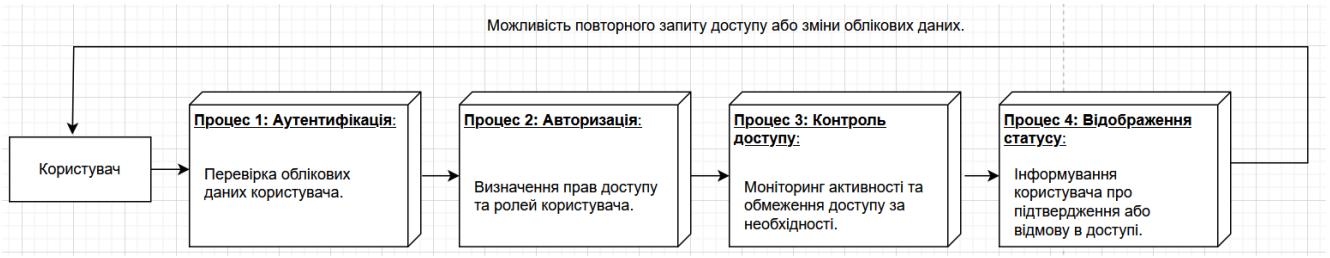


Рис. 2.5. Декомпозиція процесу Управління Доступом.

Запит користувача доступу до системи**:** Початковий етап, де користувач намагається увійти в систему.

1. Аутентифікація: Перевірка облікових даних для підтвердження особи користувача.

2. Авторизація: Визначення прав доступу та призначення відповідних ролей.

3. Контроль доступу: Моніторинг активності користувача та обмеження доступу при необхідності.

Відображення статусу: Інформування користувача про підтвердження або відмову в доступі.

2.2. Проектування інформаційного забезпечення

2.2.1. API

2.2.1.1. Визначення API

API (Application Programming Interface) — це набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення, який дозволяє різним програмам взаємодіяти між собою.

Він виступає посередником, що задає правила «спілкування» між програмами, дозволяючи їм обмінюватися даними та функціональністю без необхідності розуміти внутрішню реалізацію одна одної.

У контексті веб-додатків, API забезпечує спосіб комунікації між клієнтською частиною (наприклад, веб-браузером) та серверною частиною, яка обробляє запити та повертає необхідні дані. Коли користувач взаємодіє з веб-додатком, клієнтська

частина надсилає запити до сервера через API, який визначає, як ці запити повинні бути сформовані та які відповіді очікуються. Це дозволяє розробникам створювати модульні та масштабовані системи, де різні компоненти можуть бути розроблені та оновлені незалежно один від одного, забезпечуючи при цьому узгоджену взаємодію через чітко визначені інтерфейси.

2.2.1.2. Поняття REST

REST (Representational State Transfer) — це архітектурний стиль для створення мережевих сервісів, який визначає набір обмежень та принципів для побудови ефективних та масштабованих систем. Він був описаний і популяризований у 2000 році Роєм Філдінгом, одним із творців протоколу HTTP.

Ключові принципи REST:

1. Клієнт-серверна архітектура: Цей принцип передбачає чітке розділення між клієнтом, який відповідає за інтерфейс користувача, та сервером, який обробляє запити та зберігає дані. Таке розділення дозволяє незалежно розвивати та масштабувати обидві частини системи.

2. Відсутність стану (Statelessness): Кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для його обробки, і сервер не зберігає жодної інформації про стан між запитами. Це спрощує серверну архітектуру та покращує масштабованість.

3. Кешування: Відповіді сервера можуть бути позначені як кешовані або некешовані, що дозволяє клієнтам зберігати копії відповідей та зменшувати кількість запитів до сервера, покращуючи продуктивність системи.

4. Єдиний інтерфейс (Uniform Interface): Використання стандартизованих інтерфейсів спрощує взаємодію між компонентами системи. Це досягається через:

- Ідентифікацію ресурсів: Кожен ресурс у системі має унікальний ідентифікатор, зазвичай у вигляді URI.

- Маніпуляцію ресурсами через представлення: Клієнти взаємодіють з ресурсами через їх представлення, наприклад, у форматах JSON або XML.

- Самоописуючі повідомлення: Кожне повідомлення містить достатню інформацію для його обробки, включаючи тип даних та інструкції для обробки.

- Гіпермедіа як механізм стану застосунку (HATEOAS): Клієнти отримують необхідні посилання для подальших дій без необхідності додаткових знань про структуру API.

5. Багаторівнева система (Layered System): Архітектура може бути розділена на ієрархічні шари, де кожен шар виконує певну функцію, що сприяє підвищенню гнучкості та масштабованості системи.

6. Код за запитом (Optional Code on Demand): Сервер може надсилати виконуваний код клієнту для розширення його функціональності, хоча це не є обов'язковим принципом REST.

Дотримання цих принципів дозволяє створювати надійні, масштабовані та ефективні веб-сервіси, які легко інтегруються та підтримуються.

2.2.1.3. Структура кінцевих точок

У процесі розробки веб-додатку для бібліотечного контенту, важливо чітко визначити структуру API та відповідних кінцевих точок (див. рис. 2.9).

Рис. 2.6. Модель API

У таблицях 2.8-2.11 наведено детальний опис структури кінцевих точок, що забезпечує ефективну взаємодію з основними сутностями системи: книгами (Books), авторами (Authors), жанрами (Genres) та сесіями користувачів (Sessions).

Таблиця 2.8.

Маршрути Книг

Метод	Адреса	Пояснення
GET	/api/v1/books	Отримання списку всіх книг.
GET	/api/v1/books/{book_id}	Отримання детальної інформації про конкретну книгу.
POST	/api/v1/books	Створення нової книги.
PUT	/api/v1/books/{book_id}	Оновлення інформації про існуючу книгу.
DELETE	/api/v1/books/{book_id}	Видалення книги.

GET	/api/v1/books/{book_id}/authors	Отримання списку авторів, пов'язаних з книгою.
POST	/api/v1/books/{book_id}/authors/{author_id}	Додавання автора до книги.
DELETE	/api/v1/books/{book_id}/authors/{author_id}	Видалення автора з книги.
GET	/api/v1/books/{book_id}/genres	Отримання списку жанрів, пов'язаних з книгою.
POST	/api/v1/books/{book_id}/genres/{genre_id}	Додавання жанру до книги.
DELETE	/api/v1/books/{book_id}/genres/{genre_id}	Видалення жанру з книги.

Таблиця 2.9.

Маршрути Авторів

Метод	Адреса	Пояснення
GET	/api/v1/authors	Отримання списку всіх авторів.
GET	/api/v1/authors/{author_id}	Отримання детальної інформації про конкретного автора.
POST	/api/v1/authors	Створення нового автора.
PUT	/api/v1/authors/{author_id}	Оновлення інформації про існуючого автора.
DELETE	/api/v1/authors/{author_id}	Видалення автора.
GET	/api/v1/authors/{author_id}/books	Отримання списку книг, написаних автором.
POST	/api/v1/authors/{author_id}/books/{book_id}	Додавання книги до автора.

DELETE	/api/v1/authors/{author_id}/books/{book_id}	Видалення книги з автора.

Таблиця 2.10.

Маршрути Жанрів

Метод	Адреса	Пояснення
GET	/api/v1/genres	Отримання списку всіх жанрів.
GET	/api/v1/genres/{genre_id}	Отримання детальної інформації про конкретний жанр.
POST	/api/v1/genres	Створення нового жанру.
PUT	/api/v1/genres/{genre_id}	Оновлення інформації про існуючий жанр.
DELETE	/api/v1/genres/{genre_id}	Видалення жанру.
GET	/api/v1/genres/{genre_id}/books	Отримання списку книг, що належать до жанру.
POST	/api/v1/genres/{genre_id}/books/{book_id}	Додавання книги до жанру.
DELETE	/api/v1/genres/{genre_id}/books/{book_id}	Видалення книги з жанру.

Таблиця 2.11.

Маршрути Сесій

Метод	Адреса	Пояснення
POST	/api/v1/sessions/sign_up	Реєстрація нового користувача.
POST	/api/v1/sessions/sign_in	Вхід користувача в систему.
GET	/api/v1/sessions/current_user	Отримання інформації про поточного користувача.
PATCH	/api/v1/sessions/current_user	Оновлення інформації про поточного користувача.

DELETE	/api/v1/sessions/current_user	Видалення облікового запису поточного користувача.

Така організація кінцевих точок забезпечує чітку та логічну структуру API, що спрощує його використання та підтримку. Використання префіксу api/v1 дозволяє легко керувати версіями API та забезпечує зворотну сумісність при внесенні змін у майбутньому. Дотримання принципів RESTful архітектури, таких як використання стандартних HTTP-методів (GET, POST, PUT, DELETE) для виконання операцій над ресурсами, сприяє створенню ефективного та масштабованого веб-додатку.

2.2.2. База данних

2.2.2.1. Концептуальна модель бази даних

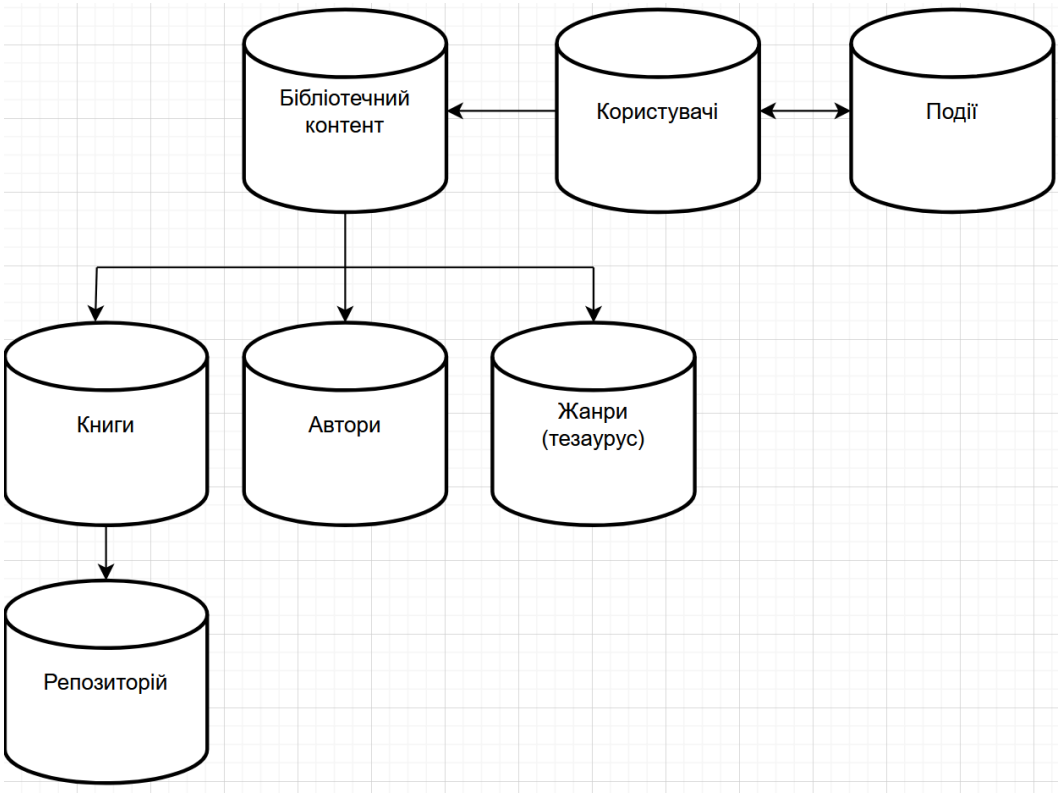


Рис 2.7. Загальна структурна схема

База даних є фундаментальною частиною будь-якої інформаційної системи, яка забезпечує збереження, обробку та управління даними. Загальна структурна

схема бази даних відображає взаємозв’язки між основними таблицями та їх атрибутами, що дозволяє ефективно організувати інформацію.

Схема бази даних побудована відповідно до принципів нормалізації, що мінімізує надлишковість даних і забезпечує логічну цілісність. Основними компонентами структури є сутності (таблиці), атрибути (поля) та зв’язки між ними. Для моделювання зв’язків використовується кілька типів взаємодії: "один до одного", "один до багатьох" і "багато до багатьох".

Концептуальна модель бази даних є абстрактним представленням інформаційних потреб предметної області, що відображає основні сутності та зв'язки між ними без деталізації атрибутів. Вона слугує основою для подальшого логічного та фізичного проєктування бази даних.

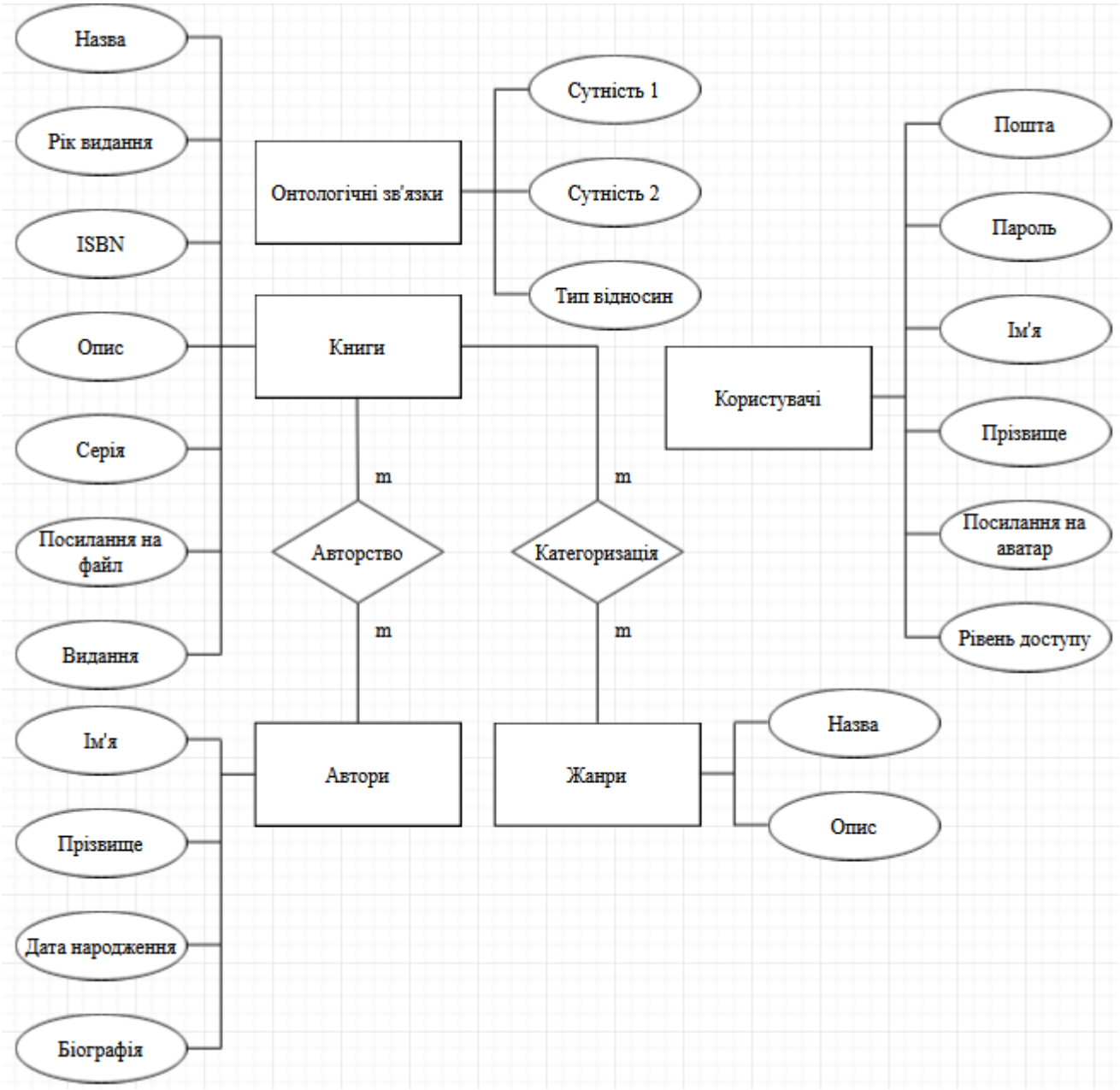


Рис. 2.8. Концептуальна модель

Сутності:

1) Книги: представляють інформацію про літературні твори, доступні в системі.
2) Автори: містять дані про осіб, які створили або брали участь у створенні книг.
3) Жанри: визначають категорії або типи літературних творів, що допомагає в їх класифікації.
4) Користувачі: включають інформацію про осіб, які взаємодіють із системою, наприклад, читачів або бібліотекарів.

5) Онтології: структури, що описують семантичні відносини та властивості між іншими сутностями, забезпечуючи глибше розуміння та зв'язність даних.

Зв'язки між сутностями:

1) Книги — Автори: зв'язок типу "багато-до-багатьох", оскільки одна книга може мати кількох авторів, а один автор може написати кілька книг.
2) Книги — Жанри: зв'язок типу "багато-до-багатьох", оскільки одна книга може належати до кількох жанрів, а один жанр може включати багато книг.
3) Онтології — інші сутності: зв'язки, що забезпечують семантичне збагачення даних, встановлюючи відносини та властивості між сутностями для покращення пошуку та аналізу інформації.

Використання онтологій для семантичного збагачення:

Онтології додають рівень семантичного збагачення, дозволяючи системі розуміти та інтерпретувати складні зв'язки між даними. Наприклад, якщо користувач цікавиться книгами певного жанру, система може рекомендувати книги авторів, які спеціалізуються на цьому жанрі, або знаходити книги, пов'язані за тематикою чи серією.

Переваги концептуальної моделі:

Чітка структура даних: визначення сутностей та їх зв'язків забезпечує зрозумілу та логічну організацію інформації.

Гнучкість: можливість легко додавати нові сутності або зв'язки без значних змін у структурі.

Покращений пошук та аналіз: семантичні зв'язки, встановлені через онтології, дозволяють здійснювати більш точний та релевантний пошук інформації.

Концептуальна модель бази даних для системи управління бібліотекою, що включає сутності "Книги", "Автори", "Жанри", "Користувачі" та "Онтології", а також визначає зв'язки між ними, забезпечує ефективну організацію та управління

інформацією. Використання онтологій для семантичного збагачення даних підвищує якість пошуку та аналізу, що сприяє покращенню обслуговування користувачів та оптимізації роботи бібліотеки.

2.2.2.2. Дата-логічна модель бази даних

Дата-логічна модель бази даних деталізує концептуальну модель, визначаючи атрибути сутностей та зв'язки між сутностями. Це забезпечує чітке уявлення про структуру даних та їх взаємозв'язки, що є основою для подальшого фізичного проєктування бази даних (див. рис. 2.8).

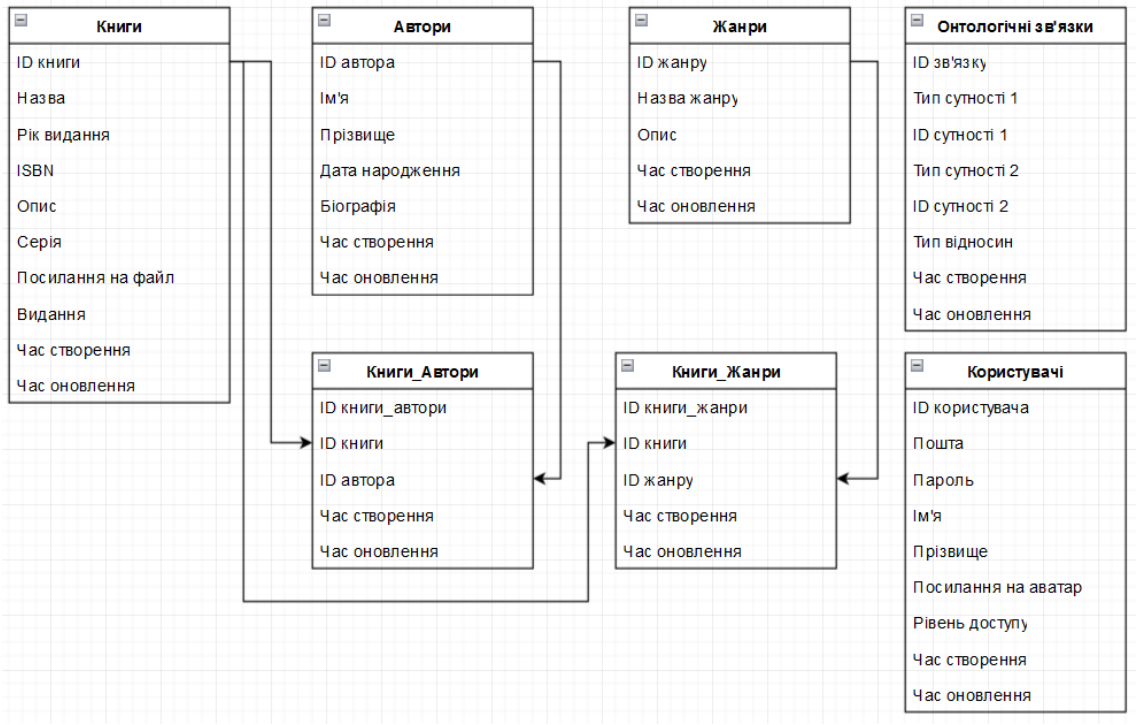


Рис. 2.9. Дата-логічна модель бази даних

Зв'язки між таблицями:

Книги — Автори: Для реалізації зв'язку "багато-до-багатьох" між книгами та авторами створюється проміжна таблиця "Автори_Книги".

Книги — Жанри: Для реалізації зв'язку "багато-до-багатьох" між книгами та жанрами створюється проміжна таблиця "Жанри_Книги".

Примітки:

1) Книги: Містить інформацію про книги, включаючи унікальний ідентифікатор (ID книги), назву, рік видання, ISBN та опис.
2) Автори: Зберігає дані про авторів, такі як ID автора, ім'я, прізвище, дата народження та біографія.
3) Жанри: Включає інформацію про жанри з ID жанру, назвою жанру та описом.
4) Авторство: Проміжна таблиця для зв'язку багато-до-багатьох між книгами та авторами, містить ID книги та ID автора.
5) Категоризація: Проміжна таблиця для зв'язку багато-до-багатьох між книгами та жанрами, містить ID книги та ID жанру.
6) Онтологічні зв'язки: Таблиця для зберігання семантичних відносин між різними сутностями, включає ID зв'язку, типи та ідентифікатори пов'язаних сутностей, а також тип відносин.

Ця логічна модель бази даних забезпечує структуроване та ефективне зберігання інформації про книги, авторів, жанри, користувачів та онтології, а також їх взаємозв'язки, що є критично важливим для функціонування інформаційної системи.

2.2.2.3 Фізична модель бази даних

Фізична модель бази даних визначає, як дані будуть організовані та збережені на рівні конкретної СУБД. Вона враховує технічні особливості, такі як типи даних, обмеження, індекси та налаштування продуктивності. Ця модель є результатом деталізації логічної моделі та включає в себе структуру таблиць, визначення зв’язків між ними, а також додаткові налаштування для оптимізації роботи бази даних.

Структура таблиць і типи даних наведені у табл. 2.1-2.7

Таблиця 2.1.

Таблиця Книги

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор книги
назва	VARCHAR	255		Назва книги
рік_видання	INTEGER	4 байти		Рік видання книги
isbn	CHAR	13	UNIQUE	Міжнародний стандартний книжковий номер
опис	TEXT	-		Опис книги
серія	VARCHAR	255		Назва серії, до якої належить книга
посилання_на_файл	VARCHAR	255		URL або шлях до файлу книги
видання	VARCHAR	255		Інформація про видання книги
час_створення	TIMESTAMP	-		Час створення запису
час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Таблиця 2.2.

Таблиця Автори

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор автора

ім'я	VARCHAR	100		Ім'я автора
прізвище	VARCHAR	100		Прізвище автора
дата_народження	DATE	-		Дата народження автора
біографія	TEXT	-		Біографія автора
час_створення	TIMESTAMP	-		Час створення запису
час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Таблиця 2.3.

Таблиця Жанри

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор жанру
назва	VARCHAR	100		Назва жанру
опис	TEXT	-		Опис жанру
час_створення	TIMESTAMP	-		Час створення запису
час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Таблиця 2.4.

Таблиця Авторство

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор запису
книга_id	INTEGER	4 байти	FK	Посилання на id книги

автор_id	INTEGER	4 байти	FK	Посилання на id автора
час_створення	TIMESTAMP	-		Час створення запису
час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Таблиця 2.5.

Таблиця Категоризація

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор запису
книга_id	INTEGER	4 байти	FK	Посилання на id книги
жанр_id	INTEGER	4 байти	FK	Посилання на id жанру
час_створення	TIMESTAMP	-		Час створення запису
час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Таблиця 2.6.

Таблиця Онтологічні зв’язки

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор зв'язку
тип_сутності_1	VARCHAR	50		Тип першої сутності

сутність_1_id	INTEGER	4 байти		Ідентифікатор першої сутності
тип_сутності_2	VARCHAR	50		Час створення запису
сутність_2_id	INTEGER	4		Тип другої сутності
тип_відносини	VARCHAR	100		Опис типу відносин між сутностями
час_створення	TIMESTAMP	-		Час створення запису
час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Таблиця 2.7.

Таблиця Користувачі

Назва поля	Тип даних	Розмір поля	Тип ключа	Коментарі
id	SERIAL	4 байти	PK	Унікальний ідентифікатор користувача
пошта	VARCHAR	255	UNIQUE	Електронна адреса користувача
пароль	VARCHAR	255		Хеш пароля користувача
ім'я	VARCHAR	100		Ім'я користувача
прізвище	VARCHAR	100		Прізвище користувача
посилання_на_аватар	VARCHAR	255		URL або шлях до аватару користувача
рівень_доступу	INTEGER	4 байти		Рівень доступу користувача
час_створення	TIMESTAMP	-		Час створення запису

час_оновлення	TIMESTAMP	-		Час останнього оновлення запису

Переваги фізичної моделі:

1) Забезпечує ефективне використання ресурсів СУБД.
2) Гарантує цілісність і послідовність даних.
3) Спрощує адміністрування та підтримку бази даних завдяки чіткій структурі та використанню обмежень.

Фізична модель дозволяє ефективно реалізувати всі вимоги до даних та зв’язків, забезпечуючи продуктивність та масштабованість системи.

2.3. Теоретико-множинна модель інформаційного забезпечення

В даному розділі описується формалізація для підсистеми управління бібліотечним контентом на основі онтологічного підходу. Структуру підсистеми можна представити наступним чином:

- **X** – множина вхідних даних:
  - (X = {x_1, x_2, x_3, x_4}), де:
    - (x_1) – дані про нові книги;
    - (x_2) – інформація про авторів;
    - (x_3) – жанрова класифікація;
    - (x_4) – запити користувачів.
- **Y** – множина вихідних даних:
  - (Y = {y_1, y_2, y_3, y_4}), де:
    - (y_1) – оновлена база даних;
    - (y_2) – результати пошуку;
    - (y_3) – підтвердження оновлення онтології;
    - (y_4) – доступ користувачів.
- **Z** – множина станів:

- (Z = {z_0, z_1, z_2, z_3, z_4, z_5}), де:
  - (z_0) – початковий стан підсистеми;
  - (z_1) – очікування введення даних;
  - (z_2) – перевірка та структурування даних;
  - (z_3) – пошук та обробка запитів;
  - (z_4) – оновлення онтології;
  - (z_5) – підтвердження доступу.

- Функції переходів підсистеми:
  - (**D** = {D_1, D_2, D_3, D_4, D_5}), де:
    - (D_1) – запит на введення даних;
    - (D_2) – передача даних для перевірки;
    - (D_3) – передача даних на пошук;
    - (D_4) – передача результатів оновлення;
    - (D_5) – підтвердження доступу.

Для опису роботи системи використовують табличне представлення у вигляді матриць переходів та виходів:

Матриця переходів**:**

Вхідні дані	(z_0)	(z_1)	(z_2)	(z_3)	(z_4)	(z_5)
(x_1)	(z_1)					
(x_2)		(z_2)				
(x_3)			(z_3)			
(x_4)				(z_4)		
Результат					(z_5)	(z_0)

Ця матриця відображає можливі переходи між станами системи на основі вхідних даних.

- Функції виходів:
    - F1 – Збір даних про нові книги.
    - F2 – Структуризація даних.
    - F3 – Оновлення бази даних.
    - F4 – Обробка запитів.
    - F5 – Оновлення онтології.
    - F6 – Підтвердження доступу.

Матриця виходів:

Вхідні дані	(y_1)	(y_2)	(y_3)	(y_4)
(x_1)	F1			
(x_2)		F2		
(x_3)			F3	
(x_4)				F4
Результат	F5	F6		

2.4. Побудова та аналіз концептуальної імітаційної моделі засобами мереж Петрі

2.4.1 Побудова моделі

Мережі Петрі дозволяють представити систему графічно, що зручно для когнітивного сприйняття інформації, а також абстрагуватися від фізичного представлення моделі і розглядати її на більш формальному рівні. Це також допомагає у програмній реалізації імітаційної моделі.

Мережа Петрі визначається наступними параметрами: (N = (P, T, I, O, M)), де:

- P – множина станів системи (позиції);
- T – множина переходів;
- I – множина вхідних функцій;

- O – множина вихідних функцій;
- M – маркування.

Для побудови імітаційної моделі процесу за принципом «причина-наслідок» засобами мереж Петрі, на підставі проведеного аналізу, виконаємо наступні кроки:

Таблиця 2.12.

Таблиця станів та переходів системи:

Стани	Призначення	Переходи	Призначення
P1	Очікування введення даних	t1	Починається введення даних
P2	Перевірка даних	t2	Завершення введення даних
P3	Структурування даних	t3	Завершення структурування даних
P4	Пошук та обробка запитів	t4	Завершення обробки запитів
P5	Оновлення онтології	t5	Початок оновлення онтології
P6	Підтвердження доступу	t6	Початок перевірки доступу
P7	Очікування результатів перевірки	t7	Результат перевірки
P8	Результат отримано		

Ця таблиця відображає послідовність станів і переходів у системі управління бібліотечним контентом.

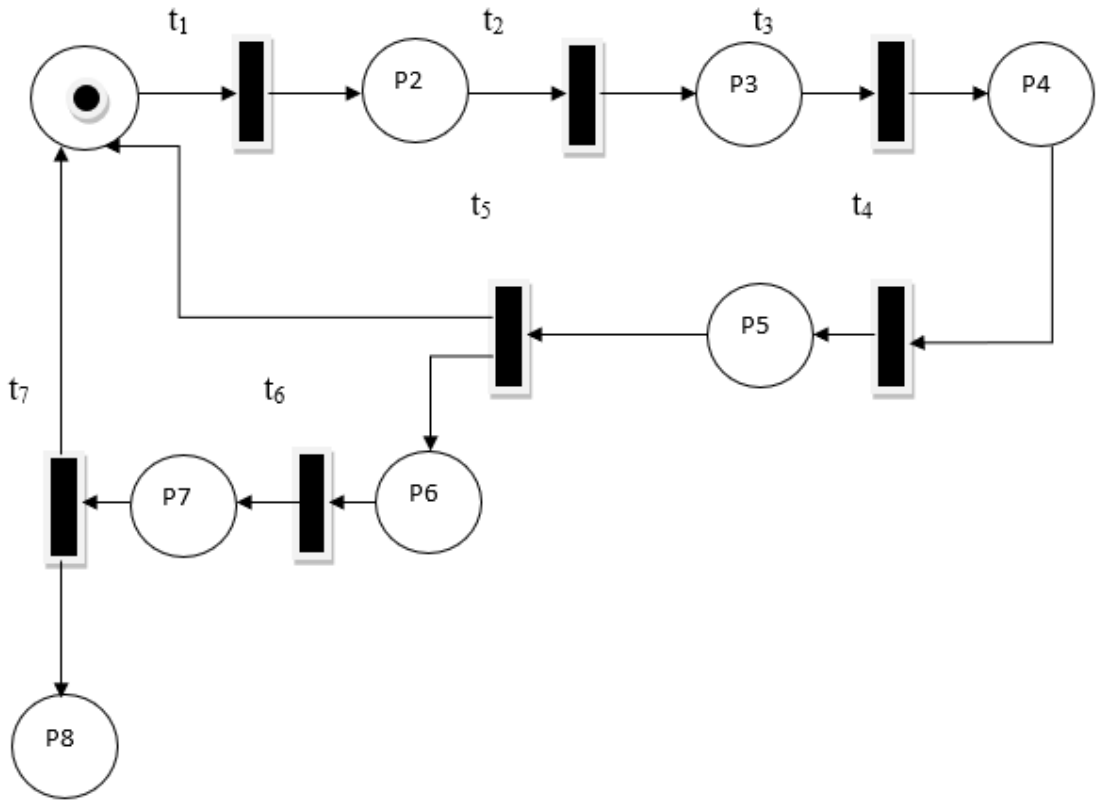


Рис. 2.10. Мережа Петрі для системи

Матриця інцидентності для мережі Петрі D-:

1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0

Матриця інцидентності для мережі Петрі D+:

0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0

0	0	0	0	1	0	0	0
1	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	1

Результуюча D:

-1	1	0	0	0	0	0	0
0	-1	1	0	0	0	0	0
0	0	-1	1	0	0	0	0
0	0	0	-1	1	0	0	0
1	0	0	0	-1	1	0	0
0	0	0	0	0	-1	1	0
1	0	0	0	0	0	-1	1

Початковою розміткою є M = (1, 0, 0, 0, 0, 0, 0, 0)

2.4.2 Аналіз моделі

Для аналізу моделі відносно її властивостей було побудовано дерево досяжності, яке представляє можливі стани системи та переходи між ними.

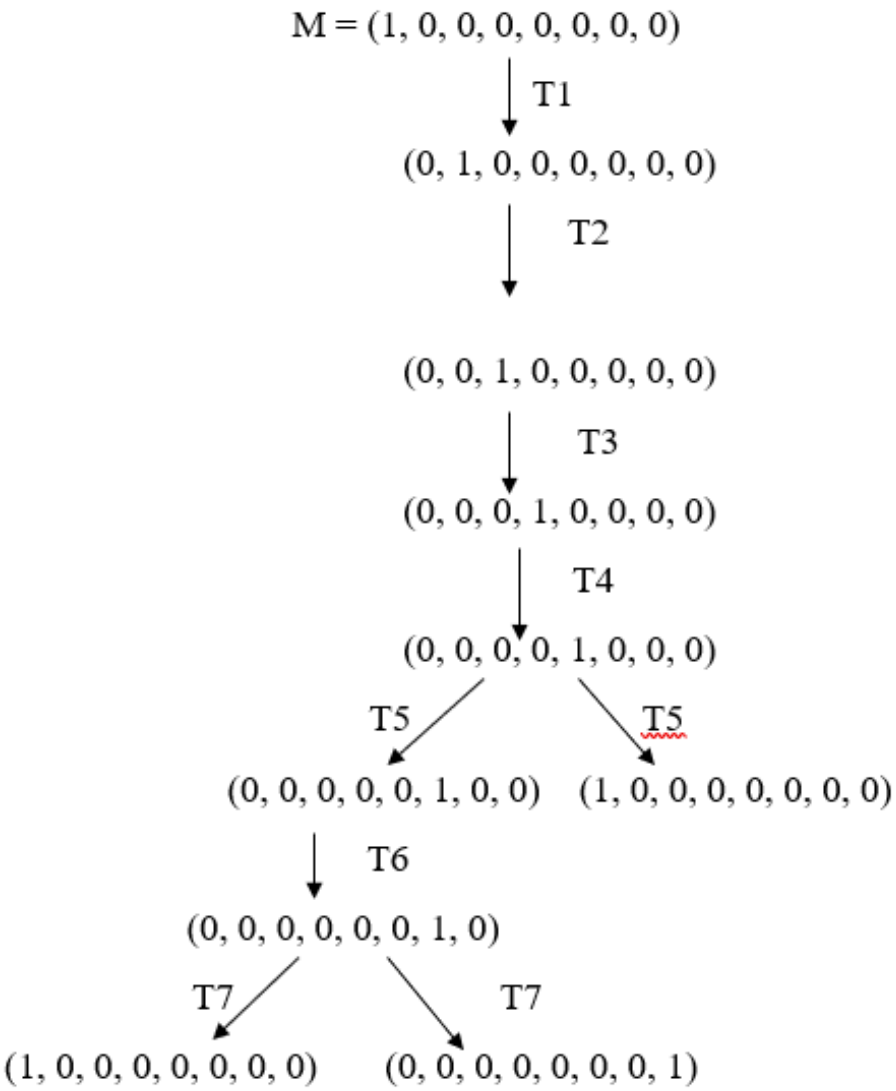


Рис. 2.11. Дерево досяжності.

Проаналізувавши побудовану модель на основі мереж Петрі відносно основних властивостей, можна зробити кілька висновків:

- Обмеженість: Кількість маркерів у кожній позиції не перевищує деякого числа n, у цій мережі вона обмежена одиницею.
- Безпека: Оскільки кількість маркерів у всіх позиціях мережі ніколи не перевищує 1, мережу можна вважати безпечною.
- Збереженість: Сума маркерів по всіх позиціях залишається строго постійною під час виконання мережі, а кількість входів дорівнює кількості виходів.
- Досяжність: Мережа може переходити з одного стану в інший, забезпечуючи повну функціональність.

3. РОЗРОБКА СИСТЕМИ ЕЛЕКТРОННОЇ БІБЛІОТЕКИ

3.1. Вибір стеку технологій

3.1.1 Мова програмування Python

Python — інтерпретована, динамічно типізована мова, що підтримує мультипарадигмове програмування (об’єктно-орієнтоване, процедурне та функціональне) і відзначається дуже широким набором стандартних та сторонніх бібліотек.

З виходом версії 3.13 було реалізовано низку оптимізацій продуктивності, серед яких попередній експериментальний JIT-компілятор та поліпшення механізму викликів методів для примітивних типів, що забезпечує приріст швидкодії до 10–20 % порівняно з Python 3.10.

Крім цього, велика спільнота розробників та мільйони пакетів на PyPI гарантують швидке знаходження готових рішень і стабільну підтримку критичних компонентів (ORM, бібліотеки для роботи з HTTP, криптографії тощо).

3.1.2 Фреймворк FastAPI

FastAPI — сучасний асинхронний веб-фреймворк, побудований на базі Starlette і Pydantic, що забезпечує автоматичну валідацію даних та генерацію документів OpenAPI/Swagger при мінімальній конфігурації.

Завдяки використанню Python type hints, FastAPI генерує клієнтський код і документацію «з коробки», скорочуючи час на тестування та інтеграцію. У тестах продуктивності (TechEmpower Benchmarks) поєднання Uvicorn + FastAPI показало одні з найвищих результатів серед Python-фреймворків, поступаючись лише чистому ASGI-серверу ©.

Широке застосування FastAPI у продуктивних рішеннях (Netflix, Uber, Microsoft) підтверджує його здатність витримувати високі навантаження та забезпечувати масштабованість системи.

3.1.3 Протокол REST

REST (Representational State Transfer) — архітектурний стиль веб-сервісів, сформульований у дисертації Роя Філдінга в 2000 р., що базується на шести ключових обмеженнях: уніфікований інтерфейс, клієнт-сервер, безстанова взаємодія, кешування, багаторівнева архітектура та код за запитом (опційно).

Застосування REST забезпечує чітку організацію ресурсів через URL, стандартизовані HTTP-методи (GET, POST, PUT, DELETE) та коди відповідей, що полегшує розробку та підтримку клієнтських і серверних компонентів, а також дозволяє використовувати проксі, балансувальники та кеші на рівні HTTP.

3.1.4 Авторизація за JWT

JSON Web Token (JWT) — компактний формат представлення заяв (claims), закодований у трьох частинах (header, payload, signature) та підписаний HMAC чи асиметричною криптографією згідно з RFC 7519.

Статичність (stateless) JWT-підходу дозволяє не зберігати сесійні дані на сервері: всю інформацію про користувача та час дії токена несе сам JWT, що знижує навантаження на БД і підвищує відмовостійкість. При цьому дотримання рекомендацій з безпеки (коректний вибір алгоритму підпису, термін життя токена, використання HTTPS) запобігає загрозам replay-атак та викраденню токенів.

3.1.5 Система управління базами даних PostgreSQL

PostgreSQL — відкрита об’єктно-реляційна СУБД із понад 35-річною історією, відома своїми властивостями ACID (Atomicity, Consistency, Isolation, Durability) і реалізацією MVCC (Multi-Version Concurrency Control).

Підтримка складних типів даних, зокрема JSONB, повнотекстового пошуку, гнучкого індексування (GIN, GiST), партиціювання та розширюваності через модулі (PostGIS, pg_trgm) робить PostgreSQL ідеальною платформою для зберігання бібліотечних метаданих і контенту.

3.1.6 Управління пакетами Poetry

Poetry — сучасний менеджер залежностей і пакувальник Python-проєктів, що об’єднує уніфіковане налаштування через *pyproject.toml* та детерміністичне блокування версій у *poetry.lock*.

Поєднання команд для керування віртуальним середовищем, автоматичного розв’язання конфліктів залежностей і публікації пакетів знижує ризики «dependency hell» та спрощує CI/CD процес.

3.1.7 Фреймворк тестування pytest

Pytest — гнучкий та розширюваний фреймворк для тестування Python-коду із підтримкою параметризації, фікстур та великої екосистеми плагінів.

Фікстури (fixtures) забезпечують модульність і повторне використання кроків підготовки/очищення середовища, а синтаксис assert з інтелектуальною інтроспекцією повідомляє про точну причину падіння тесту.

3.1.8 Порівняльна характеристика обраних технологій

У таблиці 3.1 здійснено систематизацію семи ключових інструментів проекту за їхніми технічними особливостями та практичними перевагами для забезпечення ефективної розробки, масштабованості, безпеки й підтримуваності рішення, наведено в таблиці.

Таблиця 3.1

Технологія	Основні характеристики	Переваги для проєкту
Python 3.13	JIT-компіляція (експ.), GIL-альтерн., поліпшені методи виклику	Швидкість розробки, велика спільнота, модернізація синтаксису
FastAPI	Асинхронна обробка, OpenAPI/Swagger, Pydantic, Uvicorn	Миттєве генерування документації, висока пропускна здатність
REST	Уніфікований інтерфейс, безстанова взаємодія, кешування	Легкість інтеграції, масштабованість, прозорі контракти
JWT	Статичні токени, підпис HMAC/RSA, URL-безпечність	Відмова від збереження сесій, відмовостійкість
PostgreSQL	ACID, MVCC, JSONB, індекси GIN/GiST, реплікація	Надійність даних, гнучкість схеми, висока доступність
Poetry	Єдиний конфігураційний файл, блокування версій, віртуальне середовище	Консистентність збірки, простота CI/CD
pytest	Параметризація, фікстури, плагіни, інтелектуальні asserts	Легке написання тестів, масштабованість тестового покриття

3.1.9 Бібліотеки та пакети використані для розробки

Основні залежності:

Fastapi - сучасний, високопродуктивний веб-фреймворк для створення REST-API на Python із підтримкою асинхронності, автогенерацією документації OpenAPI/Swagger та валідацією даних через type hints.

Sqlalchemy - гнучкий ORM та SQL-абстракція для Python, що складається з Core (SQL Expression Language) та ORM-рівня. Дозволяє будувати запити в Python-подібному синтаксисі та керувати моделями даних.

Sqlalchemy-utils - надбудова над SQLAlchemy, що надає додаткові утиліти та кастомні типи даних (наприклад, EmailType, UUIDType, JSONType), а також допоміжні функції для керування схемами та міграціями.

Psycopg2 - найпопулярніший адаптер PostgreSQL для Python, що реалізує DB API 2.0, підтримує потокобезпеку та ефективну роботу з великими обсягами даних.

Alembic - інструмент для версійованих міграцій бази даних, від автора SQLAlchemy. Дозволяє генерувати та застосовувати ALTER-інструкції для зміни структури таблиць у синхронному режимі.

Pyjwt - бібліотека для створення, підпису та валідації JSON Web Token (JWT) відповідно до RFC 7519. Підтримує HMAC та асиметричні алгоритми для підпису.

Passlib[bcrypt] - фреймворк для безпечного хешування паролів із реалізацією понад 30 алгоритмів (у тому числі bcrypt, argon2). Забезпечує уніфікований інтерфейс для управління хешами.

Pydantic-settings - модуль для керування налаштуваннями застосунку на базі Pydantic. Дозволяє завантажувати конфігурацію з оточення, файлів .env та інших джерел з автоматичною валідацією.

Starlette-admin - розширення для Starlette/FastAPI, що надає готовий адміністративний інтерфейс (CRUD-панель) з можливістю кастомізації та плагінів.

Itsdangerous - Набір утиліт для захищеної передачі даних у небезпечні середовища. Використовується для підпису й перевірки токенів (наприклад, для confirm-посилань) із підтримкою таймштампів.

Девелоперські залежності:

Black - «безкомпромісний» форматувальник коду для Python. Автоматично приводить код до єдиного стилю, знижуючи тривалість код-рев'ю з приводу форматування.

Pytest - фреймворк для автоматизованого тестування Python-коду з підтримкою фікстур, параметризації та плагінів. Забезпечує простий синтаксис assert з детальною інспекцією помилок.

Pytest-cov - плагін до pytest для генерації звітів покриття коду. Підтримує підрахунок покриття в дочірніх процесах і інтеграцію з різними форматами звітів.

Pytest-randomly - плагін, який випадковим чином змінює порядок запуску тестів (модулі, класи, функції). Допомагає виявляти залежності між тестами і нестабільні кейси.

Httpx - асинхронний та синхронний HTTP-клієнт для Python з підтримкою HTTP/1.1, HTTP/2, timeout, retry-механізмів та уламків під'єднань. Ідеальний для інтеграційного тестування API.

Faker - бібліотека для генерації реалістичних тестових даних: імена, адреси, тексти, email тощо. Допомагає наповнювати БД і створювати сценарії тестування з реалістичними вхідними даними.

3.2. Загальна архітектура підсистеми

У цьому пункті наведено докладний опис загальної архітектури підсистеми API електронної бібліотеки, що реалізована на базі FastAPI. Підсистема спроектована з урахуванням принципів модульності, чіткого розподілу відповідальностей (Clean/Onion Architecture), REST-орієнтованого інтерфейсу, а

також інтеграції з онтологічною моделлю (OWL/Dublin Core). Такий підхід забезпечує високу масштабованість, тестованість та можливість подальшого розширення функціоналу.

3.2.1 Принципи побудови

Підсистема API побудована за принципами REST, що передбачає чітке розмежування ресурсів і операцій над ними через стандартні HTTP-методи (GET, POST, PUT, DELETE) та коди відповідей.

З метою зниження зв’язності між модулями застосовано Onion (або Clean) Architecture, коли бізнес-логіка не залежить від деталей реалізації нижчих шарів (наприклад, доступу до БД чи фреймворка).

Для керування залежностями та підвищення тестованості використовується принцип Dependency Injection, який у FastAPI реалізується через Depends.

Асинхронна обробка запитів забезпечується через підтримку async/await в FastAPI, що дозволяє реалізувати високу конкуруваність і масштабованість під високі навантаження.

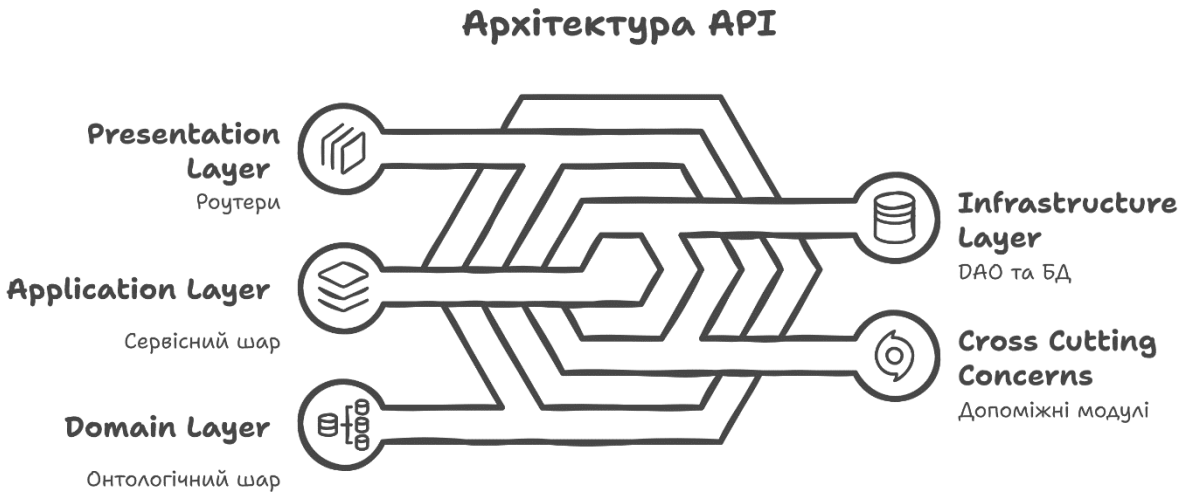


Рис. 3.1. Загальна архітектура API

3.2.2 Основні компоненти системи

3.2.2.1 Рівні архітектури

Presentation Layer (Роутери)

- Відповідає за прийом HTTP-запитів і повернення відповідей у форматі JSON/JSON-LD.
- Документується автоматично через OpenAPI/Swagger UI, вбудований у FastAPI.

Application Layer (Сервісний шар)

- Інкапсулює бізнес-логіку: валідація даних, трансформація між моделями домену та DTO.
- Реалізує послідовності операцій (транзакції), виклики до нижчих шарів та використання повторно-користованих компонентів.

Domain Layer (Онтологічний шар)

- Включає класи-домену та їх атрибути, відображені з онтології (OWL), зокрема на основі словника Dublin Core та DCMI Metadata Terms.
- Забезпечує відтворення семантичних зв’язків між об’єктами (наприклад, зв’язок “книга — автор”), керований через RDF-бібліотеку (RDFlib).

Infrastructure Layer (DAO та БД)

- Виконує фізичне збереження даних у PostgreSQL через ORM SQLAlchemy.
- Керує міграціями схеми через Alembic.

Cross-Cutting Concerns

- Логування, обробка винятків, аутентифікація/авторизація (JWT).

3.2.2.2 Опис ключових модулів

Таблиця 3.2

Модуль	Відповідальність	Технологія/Бібліотека

admin/	Реєстрація моделей для адміністративного інтерфейсу FastAPI Admin	FastAPI Admin
crud/	Реалізація базових CRUD-операцій з об’єктами бази даних	SQLAlchemy ORM
models/	Опис ORM-моделей, які відображають структуру таблиць у базі даних	SQLAlchemy ORM
routers/	Визначення REST-ендпоінтів та HTTP-маршрутів для API	FastAPI (APIRouter)
schemas/	Визначення Pydantic-схем для валідації та серіалізації даних (DTO)	Pydantic v2
services/	Бізнес-логіка, яка координує роботу між CRUD та схемами	Python, FastAPI DI (Depends)

Ці модулі відповідають рівням Presentation, Application та Infrastructure у загальній архітектурі проєкту. Таке розділення спрощує підтримку, тестування та розширення підсистеми.

3.2.3 Взаємодія компонентів

Приклад послідовності обробки запиту POST /api/v/1/books:

1. Клієнт надсилає JSON із метаданими книги відповідно до профілю Dublin Core.
2. Router викликає сервіс BooksCrud.create_book(), передаючи вже валідірований Pydantic-модельний об’єкт.
3. BookService формує доменну модель на основі OWL-класу Book, використовуючи RDFlib для встановлення семантичних зв’язків.

4. Repository зберігає новий запис у БД через SQLAlchemy і повертає ідентифікатор.
5. Router формує відповідь у форматі JSON.

3.2.4 Інтеграція з онтологічною моделлю

Онтологічний шар базується на OWL-моделі, розробленій у Protégé, де класи Book, Author, Genre відповідають RDF-класам з наборами властивостей (Dublin Core: dc:title, dc:creator, dc:subject тощо).

При створенні та читанні ресурсів API виконується двонаправлений мапінг між RDF-графом (через RDFLib) і ORM-моделями, що зберігаються в реляційній БД. Такий підхід дозволяє зберегти семантичну цілісність даних і забезпечити їхню міжнародну сумісність (JSON-LD, SPARQL-ендпоінти в майбутньому).

3.3. Програмна реалізація роутерів

Підсистема Presentation Layer у FastAPI будується на основі APIRouter — об’єкта, що групує набір «path operation» (ендпоінтів) у логічні модулі. Кожний роутер декорується HTTP-методами (@router.get(), @router.post() тощо) і використовує Dependency Injection (Depends) для отримання параметрів фільтрації, сортування, пагінації, доступу до CRUD-шару та перевірки прав користувача. Валідація вхідних і вихідних даних здійснюється через Pydantic-схеми (schemas), що задають DTO для запитів і відповідей. Завдяки такій організації коду досягається чітке розділення відповідальностей, автоматична генерація документації (OpenAPI/Swagger) та можливість масштабування.

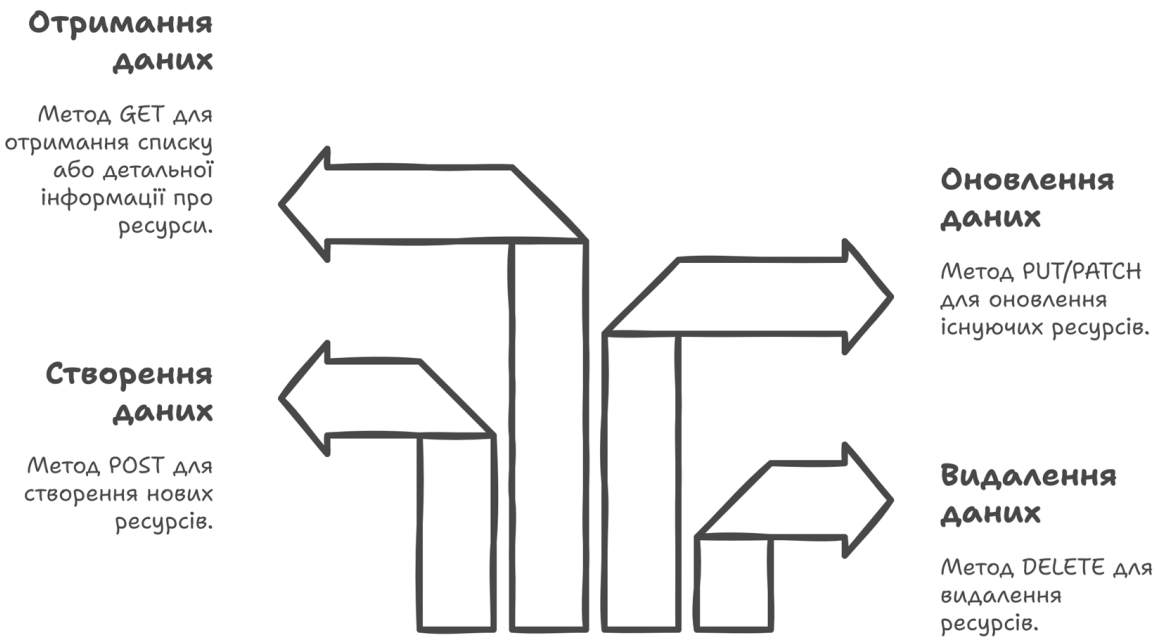


Рис. 3.2. Основні HTTP-методи

3.3.1 Принципи маршрутизації

APIRouter і модульна структура

- APIRouter — це клас із пакету FastAPI, який дозволяє групувати ендпоінти в окремі модулі (blueprints).
- Роутер створюється через router = APIRouter() і підключається до головного додатку через app.include_router(router, prefix="/authors", tags=["authors"])
- Використання префіксів і тегів полегшує навігацію по OpenAPI-документації і розмежовує функціональні блоки.

Path Operation Decorators

Кожний роутер-метод декорується одним із HTTP-декораторів: @router.get(), @router.post(), @router.patch(), @router.delete().

Параметри декоратора:

- response_model — Pydantic-схема для автоматичної валідації та серіалізації відповіді.

- status_code — встановлює HTTP-код відповіді (наприклад, 201 для успішного створення ресурсу).
- responses — додаткові можливі коди відповідей і їхні схеми/опис.

Dependency Injection (Depends)

FastAPI аналізує сигнатуру функції та «впорскує» необхідні об’єкти, параметри чи сервіси через Depends(...).

У коді для кожного ендпоінта інжектяться:

- служби фільтрації (author_search_dependency, book_search_dependency), що формують SQL-фільтри.
- параметри сортування (AuthorSortingSchema, BookSortingSchema) — Pydantic-схеми для контролю порядку.
- параметри пагінації (PaginationParams).
- CRUD-об’єкти (get_authors_crud, get_books_crud), які інкапсулюють доступ до БД.
- поточний користувач (get_librarian_user) для перевірки ролей і прав.

Валідація запитів і відповідей

Pydantic-схеми (schemas) описують структуру очікуваних JSON-об’єктів у запитах і відповіді.

Некоректні дані автоматично породжують HTTP 422 (Unprocessable Entity), додаткові помилки (404, 400, 401) задаються в responses.

Документація та тестування

FastAPI автоматично генерує OpenAPI/Swagger UI на /docs та ReDoc на /redoc.

Завдяки чіткому розподілу залежностей і шари CRUD можна легко писати unit-тести, перевизначаючи залежності через app.dependency_overrides.

3.3.2 Практична реалізація

3.3.2.1 Ініціалізація роутера

```
from fastapi import APIRouter
router = APIRouter()
```

Створюю окремий екземпляр APIRouter для кожного модуля (автори, книги, жанри, користувачі).

3.3.2.2 Приклад GET-ендпоінта зі складними залежностями

```
@router.get(
    "/",
    response_model=PaginatedResponse[AuthorSchema],
    responses=filtering_validation_error_response(),
)
async def get_authors(
    filters: dict = Depends(author_search_dependency),
    sorting_params: AuthorSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: AuthorsCrud = Depends(get_authors_crud),
):
    return crud.get_authors(
        filters=filters,
        sorting_params=sorting_params,
        pagination=pagination
    )
```

- @router.get("/") — шлях /authors/ (приєднується з префіксом).
- response_model=PaginatedResponse[AuthorSchema] — автоматичне серіалізування списку авторів з метаданими пагінації.

- filters, sorting_params, pagination — залежності, що збирають параметри запиту (query params), валідовані через Pydantic.
- crud = Depends(get_authors_crud) — клас AuthorsCrud, що інкапсулює доступ до БД.
- Метод повертає результат виклику crud.get_authors(...).

3.3.2.3 CRUD-операції: POST, PATCH, DELETE

POST /authors/

- Створює нового автора на основі CreateAuthorSchema.
- Потребує залежності current_user для перевірки прав бібліотекаря.
- Повертає створений ресурс із статусом 201.

PATCH /authors/{id}, DELETE /authors/{id}

- Оновлення та видалення за ідентифікатором.
- Обидва ендпоінти використовують комбінацію not_found_response(...) і invalid_authentication_responses() для документування можливих помилок.

3.3.2.4 Робота з пов'язаними ресурсами

- GET /authors/{id}/books — отримання книг певного автора із аналогічними механізмами фільтрації та пагінації.
- POST /authors/{id}/books/{book_id} та DELETE /authors/{id}/books/{book_id} — створення/видалення асоціації «автор — книга».
- Подібна логіка присутня в модулях books, genres.

3.3.2.5 Аутентифікація та управління користувачами

- Модуль routers/users.py містить ендпоінти для sign_up, sign_in, current_user з JWT-токенами.
- Використовуються окремі Pydantic-схеми (SignUpSchema, SignInSchema, UserSchema) і залежності Depends(get_current_user).

3.4. Програмна реалізація CRUD оцерацій

3.4.1 Утилітний модуль db_utils.py

3.4.1.1 Пошук сутностей

У модулі db_utils.py визначені загальні функції для отримання об’єктів із БД за ідентифікатором або атрибутом, які спрощують повторне використання коду і централізують обробку помилок:

- fetch_by_id(db_session, model, item_id, not_found_message) виконує SELECT ... WHERE id = :item_id і повертає один запис або кидає HTTPException(status_code=404) у разі відсутності.
- fetch_by_attr(db_session, model, attr, value, not_found_message) шукає запис за довільним атрибутом через getattr(model, attr) == value і теж повертає HTTP 404, якщо запис не знайдено.

3.4.1.2 Перевірка унікальності та асоціацій

- ensure_unique(db_session, model, field, value, error_message) завершує запит SELECT ... WHERE field = :value і кидає HTTPException(status_code=400) за наявності дубліката, забезпечуючи інваріант «унікальність поля».

- ensure_association_does_not_exist(db_session, model, **kwargs) перевіряє відсутність вже існуючого зв’язку в таблиці асоціацій (many-to-many), щоб уникнути дублювання.
- fetch_association(db_session, model, not_found_message, **kwargs) повертає існуючу асоціацію або кидає 404, якщо зв’язок відсутній.

Ці утиліти покривають типові операції з БД і використовують класичний підхід з Session.execute(select(...)) з ORM SQLAlchemy.

3.4.2 Об’єднана реалізація CRUD-класів

3.4.2.1 Загальні принципи реалізації

Repository Pattern: кожен CRUD-клас виступає репозиторієм для своєї моделі, із чітким інтерфейсом методів get_*, create_*, update_*, remove_*.

Транзакції: після add()/delete() викликається commit(); для створених/оновлених об’єктів — refresh().

Параметри запитів (фільтрація, сортування, пагінація) на рівні CRUD:

- Фільтрація — користувацька функція apply_filters(...).
- Сортування — apply_sorting(...).
- Пагінація — утиліта paginate(session, stmt, pagination) з LIMIT/OFFSET.

3.4.2.2 Формат CRUD класу

```
class Crud:
    def __init__(self, db: Session):
        self.db = db

    def get_item(self, filters, sorting_params, pagination):
        stmt = select(Item)
        if any(filters):
            stmt = apply_filters(stmt, filters,
item_search_fields)
```

```
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params,
item_sort_fields)
        return paginate(self.db, stmt=stmt,
pagination=pagination)

    def get_item_by_id(self, item_id: int):
        return fetch_by_id(self.db, Item, item_id, "Item not
found")

    def create_item(self, item_data):
        Item = Item(**item_data.model_dump())
        self.db.add(Item)
        self.db.commit()
        self.db.refresh(Item)
        return Item

    # update_item, remove_item аналогічно
    # get_books_of_item, create_item_book_association,
remove_item_book_association
```

- Уніфікований підхід до CRUD: однакова структура методів і назви, що спрощує навчання та підтримку.
- Централізовані утиліти (db_utils.py) гарантують консистентну обробку помилок і унікальності.
- Транзакційність через commit() і refresh() забезпечує актуальний стан ORM-об’єктів.
- Параметризована фільтрація/сортування/пагінація робить методи гнучкими під різні запити з UI.

3.4.3 Клас UsersCrud (модуль users.py)

3.4.3.1 Аутентифікація та реєстрація

- sign_up_user: перевіряє, що email унікальний через ensure_unique, хешує пароль за допомогою CryptContext із Passlib (bcrypt), створює та зберігає нового User, повертає об’єкт після commit() + refresh().
- sign_in_user: дістає користувача за email через fetch_by_attr, перевіряє пароль методом _verify_password (Passlib) і кидає HTTPException(status_code=401) у разі невідповідності.

3.4.3.2 Оновлення та видалення профілю

- update_user: отримує існуючого користувача, частково оновлює поля (включно з хешуванням нового пароля), викликає commit() + refresh().
- remove_user: просто видаляє через delete() + commit().

3.5. Програмна реалізація моделей та схем

3.5.1 ORM-моделі (SQLAlchemy)

3.5.1.1 Загальні налаштування

Усі моделі успадковуються від базового класу Base, створеного через declarative_base() — це дозволяє використовувати декларативний стиль опису таблиць.

Кожна таблиця має:

- __tablename__ — ім’я таблиці в БД.
- Колонки: Column із типами (Integer, String, Text, DateTime) та параметрами (primary_key, index, unique).
- Запуск функцій за умовчанням: default=func.now() і onupdate=func.now() для полів created_at/updated_at (автоматичне проставляння часу створення й оновлення).

3.5.1.2 Визначення моделей

Таблиця 3.3

Модель	Ключові поля та властивості	Опис зв’язків
Author	id, name, surname, year_of_birth, biography, created_at, updated_at	books = relationship("Book", secondary="book_author", back_populates="authors")
Book	id, title, description, year_of_publication, isbn (unique), series, file_link, edition, created_at, updated_at	authors = relationship("Author", secondary="book_author", back_populates="books"); genres = relationship("Genre", secondary="book_genre", back_populates="books")
Genre	id, name, description, created_at, updated_at	books = relationship("Book", secondary="book_genre", back_populates="genres")
BookAuthor	book_id (FK → books.id), author_id (FK → authors.id), created_at	проміжна many-to-many таблиця
BookGenre	book_id (FK → books.id), genre_id (FK → genres.id), created_at	проміжна many-to-many таблиця
User	id, email (unique, index), hashed_password, name, surname, avatar_link, access_level (default 0), created_at, updated_at	-

	Методи: is_user(), is_librarian(), is_admin()	

3.5.2 Pydantic-схеми (DTO)

3.5.2.1 Базова конфігурація

Всі схеми успадковуються від pydantic.BaseModel. Додаткові налаштування через model_config = ConfigDict(extra="forbid") забороняють передавати зайві поля та підвищують строгість валідації. Для полів із межами використовуються Field(..., ge=..., le=...).

Приклад:

```
class UpdateAuthorSchema(BaseModel):
    year_of_birth: int | None = Field(None, ge=1000, le=9999)
    model_config = ConfigDict(extra="forbid")
```

3.5.2.2 Опис сутностей

Author

- AuthorSchema — для відповіді: містить всі поля моделі, включно з id, created_at, updated_at.
- CreateAuthorSchema — для POST: обов’язкові name, surname, year_of_birth; biography опціонально (None за замовчуванням).
- UpdateAuthorSchema — для PATCH: всі поля опціональні, лише ті, що передані, оновлюються.

Book

Аналогічно:

- BookSchema — повне представлення книги.
- CreateBookSchema — обов’язкові title, year_of_publication, isbn (строго 13 цифр через pattern=r"^\d{13}$").

- UpdateBookSchema — часткове оновлення з додатковими валідаціями (min_length, max_length).

Genre

- GenreSchema, CreateGenreSchema, UpdateGenreSchema — без унікальності, стандартні поля name, description.

User

- SignUpSchema, SignInSchema, UserSchema, UpdateUserSchema — включають email, пароль (при реєстрації), ім’я, прізвище, avatar_link.

3.5.2.3 Сортування й пошук

Для кожної сутності введено додаткові схеми:

AuthorSortingSchema, BookSortingSchema, GenreSortingSchema — полями sort_by (перелік допустимих колонок) та sort_order ("asc"/"desc").

AuthorSearchSchema, BookSearchSchema, GenreSearchSchema — усі поля опціональні, задають оператор і значення у форматі "operator:value"; парсер формується через залежність у маршрутах (author_search_dependency, book_search_dependency).

3.5.3 Поєднання моделей і схем

Створення нових записів:

Декоратор у роутері приймає Pydantic-схему (CreateBookSchema), автоматично валідовану FastAPI, далі передає її в CRUD-метод, де створюється ORM-об’єкт:

```
book = Book(**book_data.model_dump())
```

Серіалізація відповідей:

Повернений ORM-екземпляр SQLAlchemy FastAPI серіалізує через response_model=BookSchema, перетворюючи поля у JSON.

Часткове оновлення:

Виклик model_dump(exclude_unset=True) повертає лише задані поля зі схеми Update*Schema, які застосовуються до атрибутів моделі перед commit().

3.6. Програмна реалізація міграцій бази данних та сервісів

3.6.1 Міграції бази даних

3.6.1.1 Використання Alembic

Alembic — легковаговий інструмент для управління міграціями реляційної бази даних, що інтегрується з SQLAlchemy і дозволяє створювати, версифікувати та застосовувати скрипти змін схеми.
При ініціалізації проєкту за допомогою alembic init генеруються конфігураційні файли (alembic.ini, папка versions/) та env.py, у якому визначено контекст підключення й схему автогенерації міграцій.

Флаг --autogenerate у команді alembic revision дозволяє аналізувати різницю між поточними моделями SQLAlchemy і схемою БД, автоматично створюючи заготовки для скриптів міграцій.
Поточні міграції застосовуються командою alembic upgrade head, що виконує всі ревізії до останньої.

3.6.1.2 Кастомний скрипт manage_db.py

Для спрощення управління БД у різних оточеннях (розробка, тестування) реалізовано утилітний скрипт на Python з використанням стандартного модуля argparse для CLI-інтерфейсу.

```
    parser = argparse.ArgumentParser(description="Database
management script.")
    parser.add_argument("command", choices=["create", "drop",
"migrate"], help="...")
```

```
    parser.add_argument("--use-test-db", action="store_true",
help="...")
    args = parser.parse_args()
```

Метод get_engine(use_test_db) читає URL з settings.DATABASE_URL або settings.TEST_DATABASE_URL і створює SQLAlchemy-движок через create_engine().

- create_db: перевіряє існування БД через database_exists(engine.url); у разі відсутності — викликає create_database(engine.url) з SQLAlchemy-Utils, інакше інформує, що БД уже є.
- drop_db: навпаки, видаляє БД через drop_database(engine.url), якщо вона існує.
- run_migrations: поєднує створення БД (якщо треба) та виклик зовнішньої команди subprocess.run(["alembic", "upgrade", "head"]), що гарантує синхронізацію схеми з міграціями.

Цей підхід дає єдиний інтерфейс «create / drop / migrate» для керування базою в різних середовищах та автоматизує застосування Alembic-міграцій.

3.6.2 Сервіси підсистеми

3.6.2.1 Сервіс авторизації (authorization.py)

Для аутентифікації використовується стандартний Bearer-токен у заголовку HTTP (HTTPBearer) разом із JWT.

- create_jwt_token(user_id: int) формує навантаження (payload) з полем sub (ідентифікатор користувача) та exp (час дії), після чого кодує його бібліотекою PyJWT (jwt.encode) з алгоритмом HS256.
- decode_jwt_token(token: str) виконує розшифровку та валідацію підпису, кидаючи 401 у разі прострочення (ExpiredSignatureError) або будь-яких інших помилок PyJWT.

- get_current_user (Depends) декодує токен, шукає користувача в БД за sub, повертає екземпляр User або 401, якщо користувач не знайдений.
- get_current_user_with_minimum_role(required_role) повертає залежність, яка додатково перевіряє рівень доступу (access_level) і кидає 403 при недостатніх правах.

Це забезпечує централізовану реалізацію JWT-авторизації й гнучке вбудовування перевірок ролей у роутери.

3.6.2.2 Сервіс пагінації (pagination.py)

Пагінація реалізована через функцію paginate(db: Session, stmt: select, pagination: PaginationParams), яка:

1. Обчислює offset = (page - 1) * size і limit = size,
2. Застосовує stmt.offset(offset).limit(limit) для витягу поточної сторінки,
3. Виконує основний запит і читає результати через scalars().all(),
4. Рахує загальну кількість записів бонусним підзапитом SELECT COUNT(*) FROM (оригінальний stmt) AS subq,
5. Обчислює кількість сторінок і повертає PaginatedResponse(items, total, page, size, pages).

Цей підхід гарантує коректний розрахунок пагінації навіть за складних JOIN та WHERE.

3.6.2.3 Сервіс фільтрації (search.py)

Для динамічної фільтрації запитів використовується словник filters: dict, де ключі — назви полів, а значення мають формат "operator:value".

- parse_filter(raw_value) розбиває рядок за ":", перевіряє оператор серед ALLOWED_OPERATORS =

{"eq","ne","lt","lte","gt","gte","like","ilike","in"} і повертає кортеж (operator, value) або HTTP 422 при невідомому операторі.

- apply_filters(stmt, filters, allowed_fields) для кожного фільтра будує умову через мапу operator_map (наприклад, col.ilike(val)) і додає її в WHERE запиту.

Такий механізм підтримує довільний набір полів та операторів без жорсткої прив’язки до конкретних ендпоінтів.

3.6.2.4 Сервіс сортування (sorting.py)

Сортування реалізоване простою функцією apply_sorting(stmt, sorting_params, sort_fields), яка:

1. Вибирає колонку зі словника sort_fields за ключем sort_by,
2. Додає до запиту ORDER BY asc(column) або desc(column) залежно від sort_order.

Це дозволяє гнучко керувати порядком результатів у CRUD-методах без дублювання коду.

4. ЕРГОНОМІЧНІ ПОКАЗНИКИ СИСТЕМИ ЕЛЕКТРОННОЇ БІБЛІОТЕКИ

4.1. Ергономічні показники програмних додатків

4.1.1 Поняття та складові ергономічних показників

Ергономічні показники програмного забезпечення—це кількісні та якісні характеристики, що визначають ступінь зручності, ефективності та безпечності взаємодії користувача з додатком. За ISO 9241-11 “юзабіліті” означає «ступінь, з якою система може бути використана певними користувачами для досягнення цілей з належною ефективністю, продуктивністю і задоволеністю у специфікованому контексті».

У межах цього визначення виокремлюють такі основні компоненти:

- Ефективність. Міра точності та повноти досягнення поставленої мети.
- Ефективність використання ресурсів. Відношення результату до затрачених часу та зусиль.
- Задоволеність. Суб’єктивна оцінка комфортності та приємності взаємодії.

4.1.2 Основні категорії показників

Зручність (Usability)

Зручність відображає інтуїтивність і логічність інтерфейсу. До її компонентів належать:

- Learnability—легкість навчання; характеризується часом, необхідним новачку, щоб успішно виконати базову задачу.
- Memorability—пам’ятність; здатність користувача відновити навички після перерви.
- Error rate—частота помилок; відсоток невдалих операцій від загальної кількості спроб.
- Satisfaction—оцінка користувача (наприклад, за системою SUS).

Продуктивність (Performance)

Вимірюється кількістю успішно виконаних завдань за одиницю часу та середнім часом виконання однієї операції.

Точність (Accuracy)

Визначається відсотком коректно виконаних операцій проти загальної кількості спроб; мінімізація помилок підвищує точність і безпеку роботи системи.

4.1.3 Швидкість виконання операцій (час відгуку)

Від швидкості роботи системи значною мірою залежить продуктивність та задоволеність користувача. Виділяють три критичні межі часу відгуку:

0,1 с—ілюзія миттєвого управління, необхідна для прямих маніпуляцій.

1 с—збереження безперервності мислення та навігації.

10 с—максимальний час без істотного відволікання; понад нього користувачі втрачають увагу і починають покидати сторінки.

Код для вимірювання часу відгуку:

```
import time
def measure_response(func, *args, **kwargs):
    start = time.perf_counter()
    result = func(*args, **kwargs)
    end = time.perf_counter()
    print(f"Час відгуку: {(end-start)*1000:.2f} ms")
    return result
```

4.1.4 Адаптивність та доступність інтерфейсу

Responsive-дизайн

WCAG 2.1 вимагає підтримки рефлоу контенту, тобто створення адаптивних сторінок під різні розміри екрану. Це забезпечує коректне відображення вмісту та зручність навігації.

Доступність (Accessibility)

Дотримання WCAG 2.1 (усі рівні A, AA) забезпечує доступність людям з порушеннями зору, слуху, моторики та когнітивних функцій. Основні успішні критерії:

- масштабованість тексту до 200 % без втрати функціональності.
- чіткі контрасти, семантична розмітка, клавіатурна навігація.

4.1.5 Таблиця основних ергономічних показників

Таблиця 4.1

Показник	Опис	Метрика	Формула
Ефективність	Повнота досягнення цілей	% успішних завдань	(N_success/N_total)·100
Продуктивність	Кількість завдань за одиницю часу	Завдання/хвилина	N_tasks / T_total
Точність	Мінімальна кількість помилок	% помилок	(N_errors/N_trials)·100
Learnability	Час, необхідний для першого успішного виконання базового завдання	Секунди	T_learn
Memorability	Час відновлення навичок після перерви	Секунди	T_remember
Час відгуку системи	Інтервал між дією користувача та реакцією системи	Мілісекунди	T_response
Задоволеність (SUS)	Суб’єктивна оцінка інтерфейсу за опитувальником System Usability Scale (0–100)	Бал	SUS_score

4.2 Ергономіка процесу проєктування

На етапі проєктування я дотримувався ISO 9241-210:2010, що визначає користувачо-центричний підхід як “процес, який робить системи зручними та корисними, фокусуючись на користувачах, їхніх потребах і вимогах”.

Це дозволило уже на стадії початкового аналізу закласти механізми інтерактивного зворотного зв’язку й регулярної перевірки прототипів із представниками цільових ролей (читачів, бібліотекарів, адміністраторів).

4.2.1 Принципи користувацько-центричного дизайну

Відповідно до ISO 9241-210, процес користувачо-центричного дизайну включає шість основних етапів:

1. Зрозуміння та визначення контексту використання – аналіз середовища, обладнання й завдань користувача.
2. Визначення вимог до користувача та завдань – збору даних за допомогою опитувань, інтерв’ю та спостережень.
3. Створення спрощених концептуальних моделей та прототипів – швидкі макети інтерфейсів і діаграми сценаріїв .
4. Ітеративне тестування з користувачами – перевірка Learnability і Satisfaction за ISO 9241-11.
5. Впровадження й оцінка відповідності стандартам ергономіки – перевірка за ISO 9241-110 (принципи діалогу).
6. Безперервне вдосконалення на основі зворотного зв’язку – аналіз метрик продуктивності та помилковості.

4.2.2 Врахування ролей та сценаріїв використання

Для кожної ролі (читач, бібліотекар, адміністратор) були розроблені UML-діаграми прецедентів (use-case), що демонструють ключові сценарії: пошук і

замовлення книг, управління каталогом, адміністрування системи. Такий підхід забезпечує:

- Чітку комунікацію між замовником і розробниками.
- Визначення пріоритетів функціоналу залежно від ролі.
- Зручну основу для тестових сценаріїв.

4.2.3 UML для моделювання баз даних та API

Діаграми класів і ER-моделювання

Класи UML використовувалися для проєктування структури бази даних: кожна сутність позначалася як клас із атрибутами й відносинами. ER-модель (Entity-Relationship) доповнювала UML-нотацію, акцентуючи увагу на кардинальностях зв’язків між сутностями.

Стандарти найменування в API

Для REST-ендпоїнтів було прийнято рекомендації проєкту RESTful API Design: використовувати іменники в множині, ієрархічне структурування URI та консистентний стиль snake_case або camelCase для параметрів JSON. Така єдність найменувань покращує передбачуваність і зменшує криву навчання для сторонніх інтеграторів.

4.2.4 Онтологічне моделювання OWL/RDF із Protégé

У проєктуванні онтологій застосовано Protégé – відкрите середовище для моделювання у форматі OWL 2 і RDF. Це дало можливість:

- Візуально структурувати класи ресурсів (монографія, стаття, автор) і властивості між ними.
- Використовувати вбудовані засоби перевірки консистентності через reasoner.
- Швидко адаптувати моделі за відгуками предметних експертів.

- Графічна нотація OWL робить онтологію зрозумілою як розробникам, так і бібліотечним фахівцям.

4.2.5 Консистентність та стандартизація

Дотримання єдиних правил у найменуванні таблиць та полів (англійською, зрозумілі абревіатури), структурі REST-запитів і форматі JSON забезпечує:

- Прогнозовану поведінку API.
- Легкість інтеграції з зовнішніми сервісами.
- Спрощення автоматизованого тестування.

4.3 Застосування CASE-засобів

Сучасні CASE-інструменти автоматизують багато рутинних завдань проєктування, що підвищує узгодженість артефактів і скорочує час на їх створення і редагування. Крім того, інтеграція таких засобів зі сховищами коду та сервісами спільної розробки забезпечує централізований контроль змін і прискорює процес рев'ю документації.

4.3.1 Моделювання ER-діаграм

Draw.io (diagrams.net) — безкоштовний вебзасіб для створення ER-діаграм із готовими шаблонами сутностей і зв'язків, підтримкою імпорту/експорту Visio, Lucidchart та интеграцією з хмарними сховищами (Google Drive, GitHub, GitLab тощо).

Dbdiagram.io — інструмент «діаграми як код», що використовує власну DSL (DBML) для швидкого побудування ER-діаграм через текстове описання таблиць і зв'язків; підтримує колаборацію в реальному часі та історію змін.

4.3.2 UML-моделювання

Visual Paradigm і Lucidchart забезпечують інтуїтивні інтерфейси для побудови UML-діаграм класів, прецедентів використання (use-case) та інших нотацій; обидва інструменти підтримують командну роботу й генерацію документації за моделями.

Використання UML дозволяє уніфікувати опис структури бази даних (через діаграми класів) і поведінки системи (через use-case), що підвищує зрозумілість для розробників і зацікавлених сторін.

4.3.3 Проєктування API з OpenAPI/Swagger

Swagger Editor — інтерактивний редактор для створення та перевірки OpenAPI-специфікації (OAS), що гарантує єдиний формат опису REST-ендпоїнтів, схем запитів/відповідей і дозволяє одразу генерувати документацію та клієнтські SDK.

Swagger Codegen — інструмент для автоматичної генерації серверних заглушок і клієнтських бібліотек на основі OAS-файлу, що пришвидшує інтеграцію фронтенду з бекендом і зменшує кількість помилок у форматах даних.

4.3.4 Системи контролю версій та хмарні сервіси

Git — розподілена система контролю версій, яка забезпечує ефективне відстеження змін, паралельну розробку в гілках і гарантує цілісність історії проєкту.

GitHub і GitLab — хмарні платформи для спільної розробки, що надають інструменти для рев'ю коду, автоматичного CI/CD, issue-трекінгу та хостингу артефактів документації; це сприяє прозорості процесу і швидкому реагуванню на зворотний зв'язок.

4.3.5 Переваги впровадження CASE-інструментів

Таблиця 4.2

Категорія	Переваги
Графічні моделювання	Швидке створення ERD/UML, готові шаблони, імпорт/експорт між інструментами
Опис API	Єдиний формат (OAS), генерація документації й SDK, спрощене тестування за допомогою Swagger UI
Автоматизація	Генерація SQL-схем, клієнтських/серверних заглушок, інтеграція з CI/CD
Контроль версій і колаборація	Чітка історія змін, code review, issue-трекінг, безперервна інтеграція

4.4 Ергономіка з точки зору користувача

4.4.1 Рольова модель доступу та її ергономічні переваги

Розподіл прав за ролями реалізовано за моделлю Role-Based Access Control (RBAC), яка обмежує доступ до функцій системи відповідно до ролі користувача.

RBAC забезпечує:

- Простоту інтерфейсу — кожен бачить лише ті елементи, які відносяться до його завдань, що зменшує навантаження на пам'ять і полегшує навчання (ISO 9241-110).
- Безпеку — обмеження прав виключає випадкові помилки чи некоректні зміни даних.

- Уніфікацію — однакові права для всіх користувачів ролі знижують потребу в індивідуальних налаштуваннях і роблять систему передбачуваною.

4.4.2 Ергономічні аспекти для читача

API-інтерфейс для читача надає тільки операції з отримання (GET) даних про книги, авторів та жанри, що мінімізує площу атаки й знижує складність для клієнтських застосунків.

- Пошук і фільтри реалізовано через кінцеві точки, що підтримують параметри запиту та повертають результати в стандартизованому форматі JSON, спрощуючи інтеграцію з фронтендом і сторонніми клієнтами.
- Пагінація й сортування здійснюються через limit, offset і sort параметри, що забезпечує передбачувану поведінку API – користувачі знають, як отримати наступну сторінку чи змінити порядок записів.
- Метадані відповіді (наприклад, total_count, page, per_page) відображають стан ресурсу й допомагають клієнту відобразити індикатори прогресу завантаження чи “хлібні крихти” всередині власного UI.

4.4.3 Ергономічні аспекти API для бібліотекаря

Бібліотекарі мають доступ до повного CRUD-функціоналу через методи POST, PUT/PATCH і DELETE.

- Управління метаданими: кінцеві точки /books (POST/PUT) та /authors підтримують валідацію полів (ISBN, дати) на рівні схеми (JSON Schema), що запобігає некоректним запитам і знижує кількість помилок на стороні клієнта.
- Контекстна документація: інтеграція з OpenAPI/Swagger UI дозволяє бібліотекарю переглядати опис ресурсів і схеми запитів без додаткових інструментів, сприяючи кращому розумінню API.

4.5 Контрольний приклад

4.5.1 Books ендпоінти

Рис. 4.1. Books ендпоінти

Чітке групування за тегом v1 books дозволяє швидко знайти всі операції з книгами.

Докладні описи параметрів фільтрації із прикладами (examples) допомагають розробнику одразу зрозуміти синтаксис операторів запиту.

Стандартизовані коди відповідей (200, 422 тощо) та їх описи роблять документацію самодокументованою.

“Try it out” у Swagger UI дає змогу протестувати GET/POST без написання жодного коду.

4.5.2 Authors ендпоінти

Рис. 4.2. Authors ендпоінти

Уніфіковані URI (/authors, /authors/{id}) відповідають REST-конвенціям, знижуючи криву навчання.

Повторне використання схем через $ref у components/schemas забезпечує єдине джерело структури даних.

Захищені операції POST/PATCH/DELETE видно по значку замка в UI, що відразу інформує про необхідність авторизації.

4.5.3 Genres ендпоінти

Рис. 4.3. Genres ендпоінти

Мінімалістичний набір методів GET дозволяє читачеві API швидко зорієнтуватися в доступних операціях.

Автоматична генерація опису параметрів path робить документацію повністю самодокументованою.

Чітке розмежування ролей — відсутність POST/PUT для жанрів у “reader” UI підкреслює модель прав доступу.

4.5.4 Sessions ендпоінти

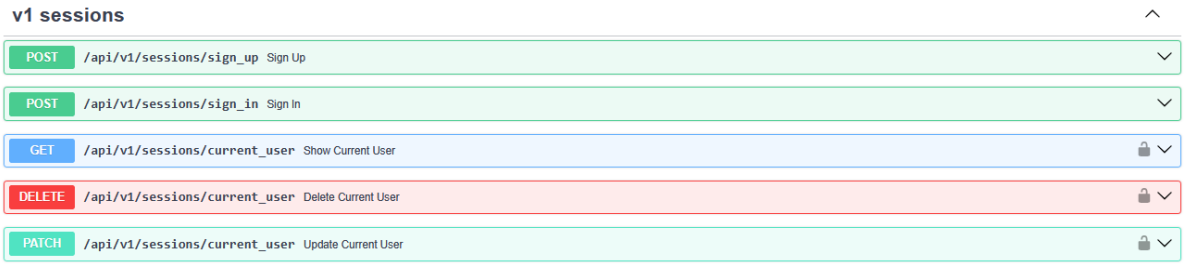


Рис. 4.4. Session ендпоінти

Логічна послідовність ендпоінтів (login → current → logout) зроблена очевидною завдяки єдиному тегу та коротким summary.

Консистентні HTTP-методи чітко відрізняють створення сесії (POST) від отримання (GET) та завершення (POST із 204).

Приклади запитів/відповідей автоматично відображаються в UI, що економить час на написання тестових клієнтів.

ВИСНОВКИ

Аналіз та дослідження проблеми

У ході аналізу проблеми встановлено, що традиційні бібліотечні системи не відповідають сучасним вимогам до управління великими обсягами даних, які постійно зростають у кількості та різноманітності. Виявлені недоліки, такі як жорсткі структури даних, відсутність семантичних зв’язків між елементами контенту та обмежені можливості інтеграції нових типів ресурсів, створюють значні труднощі для користувачів у пошуку та навігації. Ці проблеми особливо актуальні в умовах цифрової трансформації, коли бібліотеки повинні адаптуватися до нових технологій та змінюваних потреб користувачів.

Аналіз існуючих рішень, таких як Koha, Aleph, Alma та інших автоматизованих бібліотечних систем, дозволив визначити їхні сильні сторони, включаючи підтримку базових функцій каталогізації, обслуговування користувачів і управління фондами. Однак ці системи не забезпечують достатньої гнучкості для впровадження семантичного пошуку та адаптації до нових типів даних. Виявлені недоліки стали основою для формування вимог до нової системи, яка має бути орієнтована на семантичне збагачення інформації, забезпечення точного пошуку, інтеграцію мультимедійних ресурсів і підтримку адаптивного інтерфейсу.

Ключовими аспектами аналізу стали безпека, стабільність, масштабованість і адаптивність системи. Впровадження онтологічного підходу, що дозволяє формалізувати знання в певній предметній області визначено як перспективний напрямок для вирішення проблеми. Онтології забезпечують семантичне збагачення даних, що дозволяє встановлювати зв’язки між елементами контенту та покращувати релевантність результатів пошуку. Це відкриває нові можливості для створення інноваційних бібліотечних систем, які відповідатимуть сучасним вимогам інформаційного суспільства.

Проектування системи

На етапі проектування розроблено багаторівневу модель системи, яка включає концептуальну, дата-логічну та фізичну моделі бази даних. Концептуальна модель забезпечила абстрактне представлення сутностей, таких як книги, автори, жанри, користувачі та онтології, а також їх взаємозв’язків. Дата-логічна модель деталізувала атрибути сутностей і зв’язки між ними, що дозволило оптимізувати структуру даних для ефективного зберігання та обробки. Фізична модель врахувала технічні аспекти реалізації, такі як типи даних, індекси та обмеження, що забезпечило продуктивність і цілісність бази даних.

Використання онтологій стало ключовим елементом проектування, оскільки вони дозволяють додавати семантичний рівень до даних, встановлюючи зв’язки між сутностями, такими як "книга — автор" або "книга — жанр". Це забезпечує можливість більш точного та релевантного пошуку, а також інтеграцію нових концепцій без значних змін у структурі бази даних. Для моделювання процесів системи використано мережі Петрі, які дозволили графічно представити основні етапи роботи системи, такі як додавання контенту, пошук, оновлення та управління доступом. Це сприяло формалізації процесів і забезпечило передбачуваність роботи системи.

Обраний архітектурний підхід REST для проектування API забезпечив чітку організацію ресурсів і операцій над ними через стандартні HTTP методи (GET, POST, PUT, DELETE). Це дозволило створити масштабовану та гнучку систему, яка легко інтегрується з клієнтськими застосунками та забезпечує високу продуктивність навіть за умов значного навантаження.

Розробка системи

У процесі розробки обрано сучасний стек технологій, який включає Python, FastAPI, PostgreSQL, SQLAlchemy, Alembic, JWT та інші інструменти. Використання Python дозволило забезпечити швидкість розробки та доступ до великої екосистеми бібліотек, а FastAPI забезпечив високу продуктивність та автоматичне генерування документації API. PostgreSQL обрана як СУБД завдяки її

підтримці складних типів даних, таких як JSONB, та механізмів індексування, які підвищують ефективність пошуку.

Архітектура системи побудована за принципами Clean/Onion Architecture, що забезпечило чітке розділення відповідальностей між рівнями: роутери, бізнес-логіка, онтологічний шар і DAO. Реалізовані CRUD-операції дозволили ефективно працювати з ресурсами, такими як книги, автори та жанри, забезпечуючи гнучкість у додаванні, оновленні та видаленні даних. Інтеграція з онтологічною моделлю OWL через бібліотеку RDFLib дозволила зберегти семантичну цілісність даних і забезпечити їхню міжнародну сумісність.

Документація API автоматично генерується через OpenAPI/Swagger, що значно спрощує інтеграцію з клієнтськими застосунками та сторонніми сервісами. Міграції бази даних автоматизовані за допомогою Alembic, що забезпечило стабільність і контроль змін у структурі даних. Використання тестового фреймворку pytest дозволило забезпечити високий рівень покриття коду та виявити можливі помилки на ранніх етапах розробки.

Ергономічні показники системи

Ергономічні показники системи оцінені на основі стандартів ISO 9241-11 та WCAG 2.1. Основними критеріями стали зручність, продуктивність, точність, адаптивність та доступність інтерфейсу. Зручність забезпечується інтуїтивним дизайном інтерфейсу, який дозволяє користувачам легко навчатися та запам’ятовувати основні функції системи. Продуктивність вимірюється часом виконання завдань та швидкістю відгуку системи, що є критичним для забезпечення позитивного користувацького досвіду.

Адаптивність інтерфейсу реалізована через підтримку рефлоу контенту, що дозволяє коректно відображати сторінки на різних пристроях, включаючи мобільні телефони та планшети. Дотримання стандартів WCAG 2.1 забезпечило доступність системи для людей із порушеннями зору, слуху та моторики. Рольова модель доступу (RBAC) спростила інтерфейс для кожного типу користувачів, забезпечивши надійний захист даних та зручність роботи.

CASE-інструменти, такі як draw.io, dbdiagram.io та Swagger Editor, значно спростили процес проектування та документування системи. Використання систем контролю версій GitHub і GitLab забезпечило централізований контроль змін, що сприяло прозорості процесу розробки та швидкому реагуванню на зворотний зв’язок. Інтеграція з онтологічною моделлю OWL дозволила забезпечити семантичну цілісність даних, що є критично важливим для бібліотечних систем.

Таким чином, розроблена система електронної бібліотеки відповідає сучасним вимогам до ефективності, масштабованості та ергономіки. Використання онтологічного підходу дозволило інтегрувати різноманітні типи контенту, забезпечуючи високий рівень семантичного збагачення даних. Результати дослідження можуть бути використані для створення нових бібліотечних платформ, які відповідатимуть потребам інформаційного суспільства.

СПИСОК ЛІТЕРАТУРИ

1. Міністерство освіти і науки України. Методичні вказівки до виконання кваліфікаційної випускної роботи освітньо-кваліфікаційного рівня «Бакалавр» спеціальності 122 «Комп'ютерні науки». Київ: Київський національний університет будівництва та архітектури, 2024. УДК 004; 006.
2. ДСТУ ГОСТ 7.1:2006. Система стандартів з інформації, бібліотечної та видавничої справи. Бібліографічний запис. Бібліографічний опис. Чинний від 01.07.2007. Київ: Держспоживстандарт України, 2006.
3. Про бібліотеки і бібліотечну справу. Закон України, документ 32/95-ВР, чинний, редакція від 01.01.2023. URL: https://zakon.rada.gov.ua/laws/show/32/95-вр#Text.
4. Про основні засади забезпечення кібербезпеки України. Закон України, документ 2163-VIII, чинний, редакція від 01.01.2023. URL: https://zakon.rada.gov.ua/laws/show/2163-19#Text.
5. ISO 9241-11. Ergonomic requirements for office work with visual display terminals (VDTs) — Part 11: Guidance on usability. Geneva: International Organization for Standardization, 1998.
6. ISO 9241-210. Human-centred design for interactive systems. Geneva: International Organization for Standardization, 2010.
7. Dublin Core Metadata Initiative. Dublin Core Metadata Terms. [Електронний ресурс]. URL: https://dublincore.org/specifications/dublin-core/dcmi-terms/.
8. OWL 2 Web Ontology Language. W3C Recommendation. [Електронний ресурс]. URL: https://www.w3.org/TR/owl2-overview/.
9. FastAPI Documentation. Офіційна документація FastAPI. [Електронний ресурс]. URL: https://fastapi.tiangolo.com.

10. PostgreSQL Documentation. Офіційна документація PostgreSQL. [Електронний ресурс]. URL: https://www.postgresql.org/docs/.
11. Swagger Documentation. Офіційна документація Swagger/OpenAPI. [Електронний ресурс]. URL: https://swagger.io.
12. SQLAlchemy Documentation. Офіційна документація SQLAlchemy. [Електронний ресурс]. URL: https://docs.sqlalchemy.org.
13. JWT RFC 7519. JSON Web Token (JWT). [Електронний ресурс]. URL: https://www.rfc-editor.org/rfc/rfc7519.
14. Захарова О. М. Автоматизовані системи бібліотек: теорія та практика. Харків: НТУ "ХПІ", 2021. 256 с.
15. Коваленко А. В. Використання онтологій у бібліотечних системах. Київ: КПІ ім. Ігоря Сікорського, 2017. 198 с.
16. Бондаренко С. В. Основи семантичного вебу: онтології та їх застосування. Київ: КНУ ім. Тараса Шевченка, 2020. 312 с.
17. Чаплинський О. В. Проектування онтологій для семантичного вебу. Київ: Видавництво КНУ ім. Тараса Шевченка, 2022. 210 с.
18. Гутенберг М. Проект Гутенберг: цифрові бібліотеки. [Електронний ресурс]. URL: https://www.gutenberg.org.
19. Alembic Documentation. Інструмент для управління міграціями баз даних. [Електронний ресурс]. URL: https://alembic.sqlalchemy.org.
20. TechEmpower Benchmarks. Результати продуктивності веб-фреймворків. [Електронний ресурс]. URL: https://www.techempower.com/benchmarks/.
21. Passlib Documentation. Documentation for the Passlib password hashing library. [Електронний ресурс]. URL: https://passlib.readthedocs.io.
22. Protégé Documentation. OWL Ontology Editor. [Електронний ресурс]. URL: https://protege.stanford.edu.
23. RESTful API Design. Principles of RESTful API Design. [Електронний ресурс]. URL: https://restfulapi.net.

24. Ширі А. Основні напрями вивчення електронних бібліотек. International Journal on Digital Libraries, 2003, № 4(3), с. 123–136.
25. Левенштейн В. І. Алгоритм порівняння рядків. Журнал "Кібернетика", 1965, № 7(4), с. 401–407.
26. Філдінг Р. Architectural Styles and the Design of Network-based Software Architectures. Докторська дисертація. University of California, Irvine, 2000.
27. WCAG 2.1. Web Content Accessibility Guidelines (WCAG) 2.1. [Електронний ресурс]. URL: https://www.w3.org/TR/WCAG21/.
28. Pytest Documentation. Офіційна документація Pytest. [Електронний ресурс]. URL: https://docs.pytest.org.
29. Visual Paradigm Documentation. Інструмент для UML-моделювання. [Електронний ресурс]. URL: https://www.visual-paradigm.com.
30. Draw.io (diagrams.net). Інструмент для створення ER-діаграм. [Електронний ресурс]. URL: https://app.diagrams.net.

ДОДАТКИ

Додаток А

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

KYIV NATIONAL UNIVERSITY OF CONSTRUCTION AND ARCHITECTURE

CERTIFICATE

OF PARTICIPANT

INTERNATIONAL SCIENTIFIC-PRACTICAL CONFERENCE OF YOUNG SCIENTISTS

KYIV UKRAINE 29.11-01.12 2023

BUILD MASTER CLASS

Зуб Микита

(18 hours, 0.6 ECTS credits)

BMC № 2023-7-15

Rector
Dr.Sci., Professor
Petro Kulikov

Vice-Rector for Scientific Research and Innovative Development
Ph.D., Senior Research Fellow
Oleksandr Kovalchuk

BUILD-MASTER-CLASS

Ukraine, 03037
Kyiv, Povitroflotskyi ave. 31

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

KYIV NATIONAL UNIVERSITY
OF CONSTRUCTION AND ARCHITECTURE

CERTIFICATE

OF PARTICIPANT

INTERNATIONAL
SCIENTIFIC
PRACTICAL
CONFERENCE
OF YOUNG
SCIENTISTS

KYIV
UKRAINE
05-07.11
2024

BUILD
MASTER
CLASS

Mykyta ZUB

(18 hours, 0.6 ECTS credits)

№ 244

Rector
Dr.Sci., Senior Researcher
Oleksii DNIPROV

Vice-Rector for Science and Innovation
Ph.D., Senior Researcher
Oleksandr KOVALCHUK

Ukraine, Kyiv, KNUCA, Povitrianykh Syl Avenue, 31

www.economy-confer.com.ua

EC-002640

CERTIFICATE

is hereby granted to

Mykyta Zyb

for participation in the International Multidisciplinary Scientific Internet Conference
«World of scientific research. Issue 39»
with a publication on the topic: «ANALYSIS OF INFORMATION SEARCH IN WEB LIBRARIES OF FICTION»
Form of participation: remotely, duration of conference is 18 hours 0,6 ECTS credits.

The conference was held with the assistance and participation of Public Organization "Scientific Community" and Wyższa Szkoła Zarządzania i Administracji w Opolu

Organizational committee

Ternopil - Opole
March 25-26, 2025

Додаток Б

Розробка веб додатку бібліотечного контенту на основі онтологічного підходу. Частина перша. Розробка API та бази данних

Київський національний університет будівництва і архітектури

Київ 2025 рік

Доповідач: Зуб Микита Романович

Керівник: к.т.н., доц. Горда Олена Володимирівна

Обґрунтування актуальності обраної теми

Актуальність обраної теми обумовлена з викликами у сучасному інформаційному суспільстві особливо з по стійним збільшенням обсягів даних та їх різноманітністю. Це створює необхідність розробки ефективних з асобів ефективного зберігання, організації та доступу до інформаційних джерел. Особливо актуальною ця п роблема є для бібліотек, які традиційно виступають центрами знань і культурної спадщини.

Зростання вимог користувачів до доступу до інформації, інтеграції різних типів контенту та зручності пошу ку ставить перед бібліотеками завдання адаптації до нових умов. Використання онтологічного підходу для управління бібліотечним контентом дозволяє покращити вирішення цих проблеми шляхом створення гнуч ких моделей даних, що забезпечують врахування семантичного навантаження, створення класифікаторів, що покращує навігацію в пошуку необхідної інформації.

Актуальність теми також обумовлена необхідністю розробки систем, які підтримують інтеграцію різномані тних типів контенту, таких як книги, мультимедійні ресурси та наукові дані. Це сприяє підвищенню якості обслуговування користувачів, забезпечуючи їм доступ до релевантної інформації та можливість її використ ання в контексті.

При аналізі літературних джерел були виявлені основні проблеми існуючих рішень:
- ❖Відсутність семантичного збагачення даних, що ускладнює пошук і навігацію.
- ❖Обмежені можливості інтеграції нових типів контенту.
- ❖Відсутність гнучкості для адаптації до змін потреб користувачів.

Розробка web-додатку для управління бібліотечним контентом на основі онтологій є перспективним ріше нням, яке дозволить вирішити ці проблеми та забезпечити сучасний рівень організації інформації.

Об’єкт, предмет та методи дослідження

Об’єкт дослідження

Web-додаток для управління бібліотечним контентом, який використовує онтології для організації та семантичного збагачення даних.

Предмет дослідження

Методології, методи та інструментальні засоби розробки та впровадження онтологічно-орієнтованого web-додатку, включаючи архітектуру системи, функціональні компоненти та інтеграцію з існуючими бібліотечними платформами.

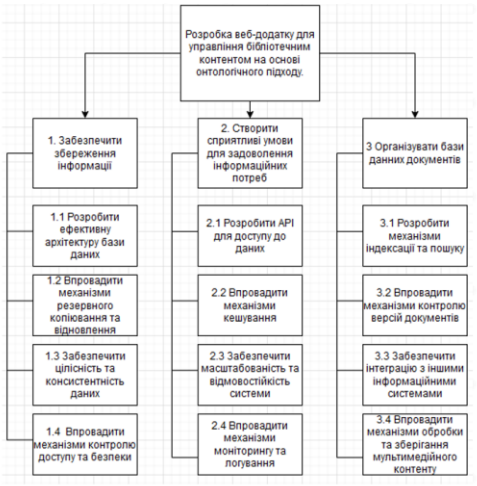
Методи дослідження

- Аналіз літературних джерел (для визначення сучасних тенденцій, викликів та перспектив використання онтологій у бібліотечних системах).
- Методи системного аналізу (для розгляду системи як єдиного цілого, виявлення її компонентів, їхніх взаємозв’язків та формалізації вимог)
- Об’єктно-орієнтована методологія аналізу та розробки програмних додатків (для виокремлення класів і об’єктів, опису їхніх властивостей і взаємодій на етапі аналізу та конструювання)
- Методи об’єктно-орієнтованого моделювання (UML-моделі: діаграми прецедентів, класів, послідовностей та активностей)
- Методи побудови онтологій (для формалізації знань предметної області)

Дерево цілей

Для відображення етапів розробки web-додатку для управління бібліотечним контентом побудовано дерево цілей, яке ієрархічно відображає шлях від загальної мети до конкретних задач.

Кожен рівень дерева цілей представляє декомпозицію основної мети на підцілі, які деталізують кроки, необхідні для її досягнення. Головна мета – розробка системи, що забезпечує ефективну організацію, семантичне збагачення та доступ до бібліотечного контенту.



Аналіз існуючих розробок та вимоги

Аналіз існуючих автоматизованих бібліотечних систем, таких як Koha, Aleph, Evergreen та інші, показав, що вони здатні виконувати базові функції управління бібліотечним контентом, включаючи каталогізацію, управління фондами та обслуговування користувачів.

Проте їхні обмеження включають:

- Відсутність семантичного збагачення даних, що ускладнює пошук і навігацію.
- Жорсткі структури даних, які не дозволяють легко інтегрувати нові типи контенту.
- Відсутність адаптивності до змінних вимог користувачів.

Використання онтологій у бібліотечних системах відкриває нові можливості:

- Семантичне збагачення даних: встановлення зв'язків між елементами контенту (наприклад, "книга — автор", "книга — жанр") для покращення релевантності пошуку.
- Гнучкість моделі даних: можливість інтеграції нових типів ресурсів без значних змін у структурі системи.
- Інтуїтивна навігація: забезпечення користувачів інструментами для розуміння контексту та взаємозв'язків між об'єктами.

На основі аналізу визначено ключові вимоги до розробки системи:

Функціональність:

- Пошук бібліотечного контенту на основі онтологій із використанням алгоритмів нечіткого пошуку.
- CRUD-операції для управління ресурсами (книги, автори, жанри).

Безпека:

- Аутентифікація та авторизація користувачів через JWT.
- Захист даних за допомогою шифрування (TLS).

Адаптивність:

- Можливість масштабування для обробки зростаючого обсягу даних.
- Підтримка багатомовності та інтеграція з глобальними інформаційними мережами.

Ергономіка:

- Інтуїтивний інтерфейс для різних категорій користувачів (читачів, бібліотекарів, адміністраторів).
- Дотримання стандартів WCAG 2.1 для забезпечення доступності.

Таким чином, онтологічний підхід є перспективним рішенням для створення сучасного, гнучкого та ефективного додатку для управління бібліотечним контентом.

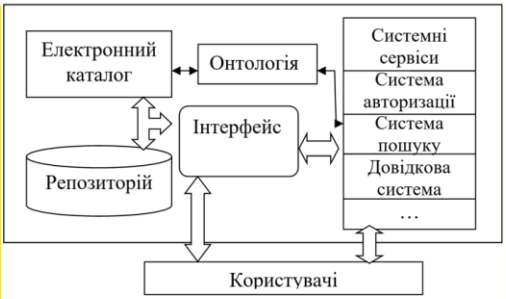
Модель «Чорна скринька» та загальна структура системи

Модель «Чорна скринька»

Демонструє принцип функціонування системи, де вхідні дані, такі як запити користувачів і оновлення контенту, проходять обробку. Система виконує пошук, структурування та управління доступом, формуючи результати пошуку та оновлену базу даних.

Загальна структура системи

Включає користувачів, пошукову систему, базу даних та інструменти управління контентом. Взаємодія між компонентами забезпечує семантичний пошук, організацію даних та моніторинг ефективності роботи системи.

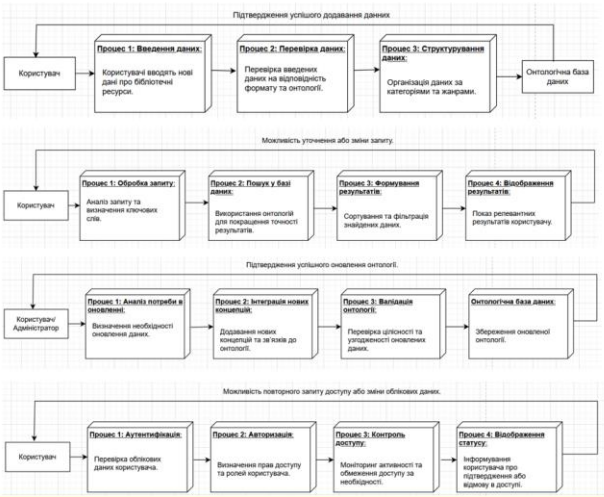


Структура системи

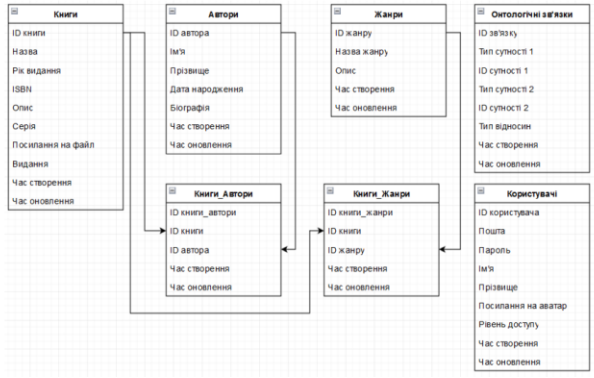
Структура системи електронної бібліотеки побудована на модульному принципі, що забезпечує чітке розділення функціональності між підсистемами. Основними підсистемами є:

- Підсистема введення даних, яка відповідає за додавання нових ресурсів до бази даних із перевіркою їхньої відповідності онтології.
- Підсистема пошуку та навігації, що реалізує семантичний пошук і фільтрацію даних на основі онтологічних зв'язків.
- Підсистема оновлення контенту, яка забезпечує актуалізацію даних та інтеграцію нових концепцій.
- Підсистема управління доступом, що контролює права користувачів і забезпечує безпеку системи.

Кожна підсистема взаємодіє з базою даних, що дозволяє централізовано керувати контентом, забезпечуючи масштабованість та інтеграцію з іншими сервісами. Завдяки використанню онтологій, система підтримує семантичне збагачення даних, що покращує пошук і навігацію.



Загальна структура бази даних



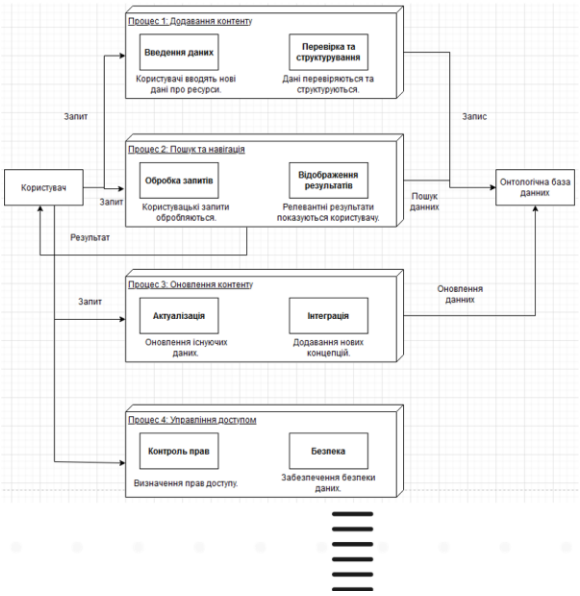
База даних системи електронної бібліотеки побудована на основі реляційної моделі з використанням принципів нормалізації. Основними сутностями є:

- Книги: містять основну інформацію про літературні твори, включаючи назву, опис, ISBN, рік видання та посилання на файл.
- Автори: зберігають дані про авторів, такі як ім'я, прізвище, дата народження та біографія.
- Жанри: класифікують книги за тематикою або типом, забезпечуючи зручну навігацію.
- Користувачі: включають інформацію про читачів, бібліотекарів та адміністраторів, а також рівень доступу.

Дата-логічна модель бази даних відображає зв'язки між сутностями, зокрема зв'язки типу «багато-до-багатьох» між книгами, авторами та жанрами. Проміжні таблиці забезпечують збереження асоціацій, таких як «Книги-Автори» та «Книги-Жанри».

Використання онтологічних зв'язків дозволяє додати семантичний рівень до даних, забезпечуючи гнучкий пошук і можливість розширення структури без значних змін. Завдяки цьому база даних підтримує інтеграцію нових типів контенту та забезпечує високу продуктивність при роботі з великими обсягами інформації.

Модель управління контентом



Управління контентом включає механізми перевірки даних на етапі додавання, що гарантує їх відповідність онтологічні й моделі. Крім того, система підтримує оновлення існуючих даних, інтеграцію нових концепцій та видалення застарілої і нформації.

Модель забезпечує гнучкість, масштабованість та зручність використання, що є важливим для ефективного функціонува ння бібліотечної системи в умовах зростаючих обсягів даних та змінних вимог користувачів.

Модель управління контентом системи електронної бібліоте ки побудована на основі чіткого розподілу функцій між комп онентами. Основними процесами є додавання, пошук, оновле ння та видалення контенту. Кожен із цих процесів реалізует ься через API, який забезпечує взаємодію між користувачами та базою даних.

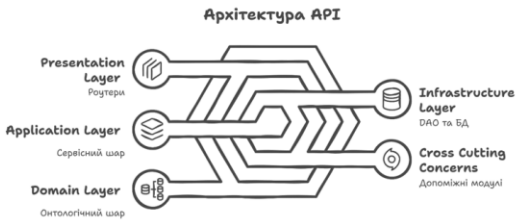
Ключовим елементом моделі є використання онтологій для структурування даних, що дозволяє встановлювати семанти чні зв'язки між книгами, авторами, жанрами та іншими сутн остями. Це забезпечує покращення релевантності пошуку, ін туїтивну навігацію та можливість розширення системи без з начних змін у її архітектурі.

Вибір стеку технологій, Загальна архітектура API

Для розробки системи електронної бібліотеки обрано сучасний стек технологій, який забезпечує високу продуктивність, гнучкі сть і масштабованість. Основні компоненти:

- Мова програмування: Python – завдяки простоті, широкому набору бібліотек і підтримці асинхронності.
- Фреймворк: FastAPI – для створення високопродуктивного REST API з автоматичною генерацією документації.
- СУБД: PostgreSQL – як надійна реляційна база даних із підтримкою JSONB та складних індексів.
- Інструменти безпеки: JWT – для аутентифікації та авторизації.
- Міграції бази даних: Alembic – для управління змінами структури даних.
- Тестування: Pytest – для забезпечення стабільності та якості коду.

Цей стек технологій дозволяє ефективно реалізувати всі вимог и до системи, включаючи семантичне збагачення даних, швидк ий пошук і високу надійність.



Архітектура API складається з кількох рівнів, що забезпечують чітке розділення функцій. Роутери приймають запити та пере дають їх на сервісний шар, який реалізує бізнес-логіку. Домен ний шар використовує онтологічну модель для семантичного збагачення даних, а інфраструктурний шар відповідає за досту п до бази даних. Така структура забезпечує модульність, гнучкі сть і масштабованість системи.

Опис ключових модулів

Система електронної бібліотеки складається з кількох ключових модулів, кожен із яких відповідає за певний аспект функціонування. Основними модулями є: модуль адміністративного інтерфейсу, модуль CRUD-операцій, модуль роутерів для обробки HTTP-запитів, модуль схем для валідації даних та модуль сервісів для реалізації бізнес-логіки.

Таблиця деталізує основні модулі, їхню відповідальність та використані технології.

Модуль	Відповідальність	Технологія/Бібліотека
admin/	Реєстрація моделей для адміністративного інтерфейсу	FastAPI Admin
crud/	Реалізація CRUD-операцій із базою даних	SQLAlchemy ORM
models/	Опис ORM-моделей для структури таблиць	SQLAlchemy ORM
routers/	Визначення REST-ендпоінтів	FastAPI (APIRouter)
schemas/	Валідація та серіалізація даних	Pydantic
services/	Бізнес-логіка та координація між модулями	Python, FastAPI DI (Depends)

Ергономіка системи електронної бібліотеки

Ергономіка системи розроблена відповідно до стандартів ISO 9241-11 та WCAG 2.1, що забезпечує зручність, доступність та ефективність використання. Інтерфейс системи є інтуїтивним, що дозволяє користувачам легко навчатися та швидко виконувати завдання.

Система підтримує адаптивний дизайн, який забезпечує коректне відображення на пристроях із різними розмірами екрану. Доступність реалізована через масштабованість тексту, контрастність елементів та клавіатурну навігацію, що робить систему зручною для людей із порушеннями зору, слуху чи моторики.

Рольова модель доступу (RBAC) спрощує взаємодію користувачів із системою, показуючи лише ті функції, які відповідають їхнім ролям (читач, бібліотекар, адміністратор). Це зменшує когнітивне навантаження та підвищує продуктивність роботи.

Ергономічні показники системи включають:

- Час відгуку системи менше 1 секунди, що забезпечує безперервність роботи.
- Високий рівень задоволеності користувачів, оцінений за шкалою SUS.
- Мінімальну кількість помилок завдяки вбудованій валідації даних.

Контрольний приклад

У системі електронної бібліотеки реалізовано набір REST-ендпоінтів для роботи з книгами та авторами, які автоматично документуються у Swagger завдяки використанню FastAPI. Ендпоінти забезпечують повний функціонал CRUD-операцій, включаючи створення, редагування, видалення та отримання даних.

Для Books:

- Ендпоінти дозволяють отримувати список книг, детальну інформацію про конкретну книгу, додавати нові книги, редагувати існуючі записи та видаляти книги.
- Також реалізовано асоціації книг із авторами та жанрами через відповідні ендпоінти.

Для Authors:

- Кінцеві точки дозволяють працювати зі списком авторів, отримувати інформацію про конкретного автора, створювати нових авторів, оновлювати їхні дані та видаляти записи.
- Додатково підтримується асоціація авторів із книгами для зручного управління взаємозв'язками.

У Swagger також представлено ендпоінти для роботи з жанрами та сесіями користувачів.

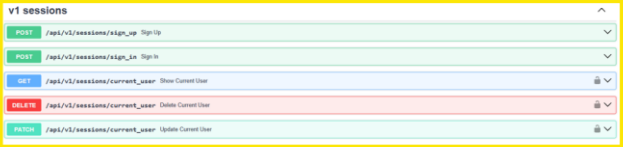
Для Genres:

- Ендпоінти дозволяють отримувати список жанрів, детальну інформацію про конкретний жанр, створювати нові жанри, редагувати існуючі записи та видаляти жанри.
- Асоціація жанрів із книгами реалізована через спеціальні кінцеві точки, що забезпечують гнучкість у роботі з контентом.

Для Sessions:

- Реалізовано ендпоінти для реєстрації нового користувача, входу в систему, отримання даних про поточного користувача, завершення сесії та інші.
- Авторизація здійснюється через JWT-токени, що гарантує безпеку та контроль доступу.

Swagger відображає всі деталі запитів та відповідей, включаючи статуси, формати даних та приклади, що робить API прозорим і зручним для використання.



Висновки

У процесі розробки системи електронної бібліотеки було вирішено низку важливих задач, спрямованих на створення сучасного та ефективного інструменту для управління бібліотечним контентом.

Аналіз існуючих рішень показав, що традиційні бібліотечні системи мають обмеження у семантичному пошуку, інтеграції нових типів контенту та адаптивності до потреб користувачів. Це стало основою для впровадження онтологічного підходу, який забезпечив семантичне збагачення даних, покращення навігації та можливість встановлення зв'язків між сутностями, такими як книги, автори та жанри.

Розроблена архітектура системи базується на принципах модульності та багаторівневості, що забезпечує її масштабованість, гнучкість і високу продуктивність. Використання сучасного стеку технологій, включаючи FastAPI, PostgreSQL та JWT, дозволило створити систему, яка відповідає вимогам безпеки, стабільності та ефективності.

Ергономічні аспекти системи реалізовані відповідно до стандартів ISO 9241-11 та WCAG 2.1, що забезпечує зручність, доступність та адаптивність для різних категорій користувачів. Рольова модель доступу спрощує взаємодію користувачів із системою, а інтуїтивний інтерфейс сприяє швидкому освоєнню функціоналу.

Таким чином, розроблена система електронної бібліотеки є сучасним рішенням, яке забезпечує ефективне управління контентом, інтеграцію різноманітних типів даних та високий рівень задоволеності користувачів.

Дякую за увагу !

Додаток В

Повний код знаходиться на ресурсі Github за посиланням https://github.com/TapTapStitch/FastAPILibraryProject

Код програми:

app/main.py

```
from fastapi import FastAPI
from fastapi.responses import RedirectResponse
from app.routers.api.v1 import authors, genres, sessions, books
from app.admin.index import admin

app = FastAPI(debug=True)
admin.mount_to(app)

app.include_router(books.router, prefix="/api/v1/books", tags=["v1 books"])
app.include_router(authors.router, prefix="/api/v1/authors", tags=["v1 authors"])
app.include_router(genres.router, prefix="/api/v1/genres", tags=["v1 genres"])
app.include_router(sessions.router, prefix="/api/v1/sessions", tags=["v1
sessions"])


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


@app.get("/", include_in_schema=False)
async def redirect_to_docs():
    return RedirectResponse(url="/docs")
```

app/config.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker


class Settings(BaseSettings):
    DATABASE_URL: str
    JWT_SECRET_KEY: str
    JWT_TOKEN_EXPIRATION: int | None = 60
```

```
    COOKIE_SECRET_KEY: str
    TEST_DATABASE_URL: str | None = None
    DEBUG: bool | None = False

    model_config = SettingsConfigDict(env_file=".env")


settings = Settings()
engine = create_engine(settings.DATABASE_URL, echo=settings.DEBUG)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()


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

app/admin/auth.py

```
from sqlalchemy.future import select
from starlette.requests import Request
from starlette.responses import Response
from starlette_admin.auth import AdminConfig, AdminUser, AuthProvider
from starlette_admin.exceptions import LoginFailed
from app.models import User
from app.config import get_db
from passlib.context import CryptContext

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


class EmailAndPasswordProvider(AuthProvider):
    async def login(
        self,
        email: str,
        password: str,
        remember_me: bool,
        request: Request,
        response: Response,
```

```
) -> Response:
    db = next(get_db())
    stmt = select(User).where(User.email == email)
    result = db.execute(stmt)
    user = result.scalar_one_or_none()

    if user is None:
        raise LoginFailed("Invalid email or password")
    if not pwd_context.verify(password, user.hashed_password):
        raise LoginFailed("Invalid email or password")
    if not user.is_admin():
        raise LoginFailed("Access denied. User is not admin")

    request.session["user_id"] = user.id
    return response

async def is_authenticated(self, request: Request) -> bool:
    user_id = request.session.get("user_id")
    if user_id is None:
        return False

    db = next(get_db())
    stmt = select(User).where(User.id == user_id)
    result = db.execute(stmt)
    user = result.scalar_one_or_none()

    if user and user.is_admin():
        request.state.user = user
        return True
    return False

def get_admin_config(self, request: Request) -> AdminConfig:
    user: User = request.state.user
    custom_app_title = f"Hello, {user.name}!"
    return AdminConfig(app_title=custom_app_title)

def get_admin_user(self, request: Request) -> AdminUser:
    user: User = request.state.user
    return AdminUser(username=user.name)

async def logout(self, request: Request, response: Response) -> Response:
```

```
        request.session.clear()
        return response
```

app/admin/index.py

```
from starlette.middleware import Middleware
from starlette.middleware.sessions import SessionMiddleware
from starlette_admin.contrib.sqla import Admin, ModelView
from app.config import engine, settings
from app.models import Book, Author, Genre, User
from app.admin.auth import EmailAndPasswordProvider, pwd_context

admin = Admin(
    engine,
    title="Library Admin Panel",
    auth_provider=EmailAndPasswordProvider(),
    middlewares=[Middleware(SessionMiddleware,
secret_key=settings.COOKIE_SECRET_KEY)],
)


class CustomModelView(ModelView):
    exclude_fields_from_create = ["created_at", "updated_at"]
    exclude_fields_from_edit = ["created_at", "updated_at"]


class UserAdmin(CustomModelView):
    async def before_create(self, request, data: dict, item: User) -> None:
        if item.hashed_password and not item.hashed_password.startswith("$"):
            item.hashed_password = pwd_context.hash(item.hashed_password)

    async def before_edit(self, request, data: dict, item: User) -> None:
        if item.hashed_password and not item.hashed_password.startswith("$"):
            item.hashed_password = pwd_context.hash(item.hashed_password)


admin.add_view(UserAdmin(User, name="Users"))
admin.add_view(CustomModelView(Book, name="Books"))
admin.add_view(CustomModelView(Author, name="Authors"))
admin.add_view(CustomModelView(Genre, name="Genres"))
```

app/crud/api/v1/books.py

```
from sqlalchemy.orm import Session
from sqlalchemy import select
from app.services.pagination import paginate
from app.models.book import Book
from app.models.author import Author
from app.models.book_author import BookAuthor
from app.models.genre import Genre
from app.models.book_genre import BookGenre
from app.schemas.api.v1.book import (
    CreateBookSchema,
    UpdateBookSchema,
    BookSortingSchema,
)
from app.schemas.api.v1.author import AuthorSortingSchema
from app.schemas.api.v1.genre import GenreSortingSchema
from app.schemas.pagination import PaginationParams
from app.services.sorting import apply_sorting
from app.services.search import apply_filters
from app.crud.shared.db_utils import (
    fetch_by_id,
    ensure_unique,
    ensure_association_does_not_exist,
    fetch_association,
)
from app.crud.api.v1.shared.sort_fields import (
    book_sort_fields,
    author_sort_fields,
    genre_sort_fields,
)
from app.crud.api.v1.shared.search_filelds import (
    book_search_fields,
    author_search_fields,
    genre_search_fields,
)


class BooksCrud:
    def __init__(self, db: Session):
        self.db = db

    def get_books(
        self,
```

```
        filters: dict,
        sorting_params: BookSortingSchema,
        pagination: PaginationParams,
    ):
        stmt = select(Book)
        if any(filters):
            stmt = apply_filters(stmt, filters, book_search_fields)
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params, book_sort_fields)
        return paginate(self.db, stmt=stmt, pagination=pagination)

    def get_book_by_id(self, book_id: int):
        return fetch_by_id(self.db, Book, book_id, "Book not found")

    def create_book(self, book_data: CreateBookSchema):
        ensure_unique(self.db, Book, "isbn", book_data.isbn, "ISBN must be
unique")
        book = Book(**book_data.model_dump())
        self.db.add(book)
        self.db.commit()
        self.db.refresh(book)
        return book

    def update_book(self, book_id: int, book_data: UpdateBookSchema):
        book = self.get_book_by_id(book_id)
        updated_data = book_data.model_dump(exclude_unset=True)

        if "isbn" in updated_data and updated_data["isbn"] != book.isbn:
            ensure_unique(
                self.db, Book, "isbn", updated_data["isbn"], "ISBN must be unique"
            )

        for field, value in updated_data.items():
            setattr(book, field, value)

        self.db.commit()
        self.db.refresh(book)
        return book

    def remove_book(self, book_id: int):
        book = self.get_book_by_id(book_id)
        self.db.delete(book)
```

```
        self.db.commit()

    def get_authors_of_book(
        self,
        book_id: int,
        filters: dict,
        sorting_params: AuthorSortingSchema,
        pagination: PaginationParams,
    ):
        self.get_book_by_id(book_id)
        stmt = (
            select(Author)
            .join(BookAuthor, Author.id == BookAuthor.author_id)
            .where(BookAuthor.book_id == book_id)
        )
        if any(filters):
            stmt = apply_filters(stmt, filters, author_search_fields)
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params, author_sort_fields)
        return paginate(self.db, stmt=stmt, pagination=pagination)

    def create_book_author_association(self, book_id: int, author_id: int):
        self.get_book_by_id(book_id)
        fetch_by_id(self.db, Author, author_id, "Author not found")
        ensure_association_does_not_exist(
            self.db, BookAuthor, book_id=book_id, author_id=author_id
        )
        self.db.add(BookAuthor(book_id=book_id, author_id=author_id))
        self.db.commit()

    def remove_book_author_association(self, book_id: int, author_id: int):
        self.get_book_by_id(book_id)
        fetch_by_id(self.db, Author, author_id, "Author not found")
        association = fetch_association(
            self.db,
            BookAuthor,
            "Association not found",
            book_id=book_id,
            author_id=author_id,
        )
        self.db.delete(association)
        self.db.commit()
```

```
def get_genres_of_book(
    self,
    book_id: int,
    filters: dict,
    sorting_params: GenreSortingSchema,
    pagination: PaginationParams,
):
    self.get_book_by_id(book_id)
    stmt = (
        select(Genre)
        .join(BookGenre, Genre.id == BookGenre.genre_id)
        .where(BookGenre.book_id == book_id)
    )
    if any(filters):
        stmt = apply_filters(stmt, filters, genre_search_fields)
    if sorting_params.sort_by:
        stmt = apply_sorting(stmt, sorting_params, genre_sort_fields)
    return paginate(self.db, stmt=stmt, pagination=pagination)

def create_book_genre_association(self, book_id: int, genre_id: int):
    self.get_book_by_id(book_id)
    fetch_by_id(self.db, Genre, genre_id, "Genre not found")
    ensure_association_does_not_exist(
        self.db, BookGenre, book_id=book_id, genre_id=genre_id
    )
    self.db.add(BookGenre(book_id=book_id, genre_id=genre_id))
    self.db.commit()

def remove_book_genre_association(self, book_id: int, genre_id: int):
    self.get_book_by_id(book_id)
    fetch_by_id(self.db, Genre, genre_id, "Genre not found")
    association = fetch_association(
        self.db,
        BookGenre,
        "Association not found",
        book_id=book_id,
        genre_id=genre_id,
    )
    self.db.delete(association)
    self.db.commit()
```

app/crud/api/v1/authors.py

```
from sqlalchemy.orm import Session
from sqlalchemy import select
from app.services.pagination import paginate
from app.models.book import Book
from app.models.author import Author
from app.models.book_author import BookAuthor
from app.schemas.api.v1.author import (
    CreateAuthorSchema,
    UpdateAuthorSchema,
    AuthorSortingSchema,
)
from app.schemas.api.v1.book import BookSortingSchema
from app.schemas.pagination import PaginationParams
from app.services.sorting import apply_sorting
from app.services.search import apply_filters
from app.crud.shared.db_utils import (
    fetch_by_id,
    ensure_association_does_not_exist,
    fetch_association,
)
from app.crud.api.v1.shared.sort_fields import book_sort_fields,
author_sort_fields
from app.crud.api.v1.shared.search_filelds import (
    book_search_fields,
    author_search_fields,
)


class AuthorsCrud:
    def __init__(self, db: Session):
        self.db = db

    def get_authors(
        self,
        filters: dict,
        sorting_params: AuthorSortingSchema,
        pagination: PaginationParams,
    ):
        stmt = select(Author)
        if any(filters):
```

```
            stmt = apply_filters(stmt, filters, author_search_fields)
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params, author_sort_fields)
        return paginate(self.db, stmt=stmt, pagination=pagination)

    def get_author_by_id(self, author_id: int):
        return fetch_by_id(self.db, Author, author_id, "Author not found")

    def create_author(self, author_data: CreateAuthorSchema):
        author = Author(**author_data.model_dump())
        self.db.add(author)
        self.db.commit()
        self.db.refresh(author)
        return author

    def update_author(self, author_id: int, author_data: UpdateAuthorSchema):
        author = self.get_author_by_id(author_id)
        updated_data = author_data.model_dump(exclude_unset=True)

        for field, value in updated_data.items():
            setattr(author, field, value)

        self.db.commit()
        self.db.refresh(author)
        return author

    def remove_author(self, author_id: int):
        author = self.get_author_by_id(author_id)
        self.db.delete(author)
        self.db.commit()

    def get_books_of_author(
        self,
        author_id: int,
        filters: dict,
        sorting_params: BookSortingSchema,
        pagination: PaginationParams,
    ):
        self.get_author_by_id(author_id)
        stmt = (
            select(Book)
            .join(BookAuthor, Book.id == BookAuthor.book_id)
```

```
            .where(BookAuthor.author_id == author_id)
        )
        if any(filters):
            stmt = apply_filters(stmt, filters, book_search_fields)
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params, book_sort_fields)
        return paginate(self.db, stmt=stmt, pagination=pagination)

    def create_author_book_association(self, author_id: int, book_id: int):
        self.get_author_by_id(author_id)
        fetch_by_id(self.db, Book, book_id, "Book not found")
        ensure_association_does_not_exist(
            self.db, BookAuthor, author_id=author_id, book_id=book_id
        )
        self.db.add(BookAuthor(author_id=author_id, book_id=book_id))
        self.db.commit()

    def remove_author_book_association(self, author_id: int, book_id: int):
        self.get_author_by_id(author_id)
        fetch_by_id(self.db, Book, book_id, "Book not found")
        association = fetch_association(
            self.db,
            BookAuthor,
            "Association not found",
            author_id=author_id,
            book_id=book_id,
        )
        self.db.delete(association)
        self.db.commit()
```

app/crud/api/v1/genres.py

```
from sqlalchemy.orm import Session
from sqlalchemy import select
from app.services.pagination import paginate
from app.models.genre import Genre
from app.models.book import Book
from app.models.book_genre import BookGenre
from app.schemas.api.v1.genre import (
    CreateGenreSchema,
    UpdateGenreSchema,
    GenreSortingSchema,
```

```
)
from app.schemas.api.v1.book import BookSortingSchema
from app.schemas.pagination import PaginationParams
from app.services.sorting import apply_sorting
from app.services.search import apply_filters
from app.crud.shared.db_utils import (
    fetch_by_id,
    ensure_association_does_not_exist,
    fetch_association,
)
from app.crud.api.v1.shared.sort_fields import book_sort_fields, genre_sort_fields
from app.crud.api.v1.shared.search_filelds import (
    book_search_fields,
    genre_search_fields,
)


class GenresCrud:
    def __init__(self, db: Session):
        self.db = db

    def get_genres(
        self,
        filters: dict,
        sorting_params: GenreSortingSchema,
        pagination: PaginationParams,
    ):
        stmt = select(Genre)
        if any(filters):
            stmt = apply_filters(stmt, filters, genre_search_fields)
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params, genre_sort_fields)
        return paginate(self.db, stmt=stmt, pagination=pagination)

    def get_genre_by_id(self, genre_id: int):
        return fetch_by_id(self.db, Genre, genre_id, "Genre not found")

    def create_genre(self, genre_data: CreateGenreSchema):
        genre = Genre(**genre_data.model_dump())
        self.db.add(genre)
        self.db.commit()
        self.db.refresh(genre)
```

```
        return genre

    def update_genre(self, genre_id: int, genre_data: UpdateGenreSchema):
        genre = self.get_genre_by_id(genre_id)
        updated_data = genre_data.model_dump(exclude_unset=True)

        for field, value in updated_data.items():
            setattr(genre, field, value)

        self.db.commit()
        self.db.refresh(genre)
        return genre

    def remove_genre(self, genre_id: int):
        genre = self.get_genre_by_id(genre_id)
        self.db.delete(genre)
        self.db.commit()

    def get_books_of_genre(
        self,
        genre_id: int,
        filters: dict,
        sorting_params: BookSortingSchema,
        pagination: PaginationParams,
    ):
        self.get_genre_by_id(genre_id)
        stmt = (
            select(Book)
            .join(BookGenre, Book.id == BookGenre.book_id)
            .where(BookGenre.genre_id == genre_id)
        )
        if any(filters):
            stmt = apply_filters(stmt, filters, book_search_fields)
        if sorting_params.sort_by:
            stmt = apply_sorting(stmt, sorting_params, book_sort_fields)
        return paginate(self.db, stmt=stmt, pagination=pagination)

    def create_genre_book_association(self, genre_id: int, book_id: int):
        self.get_genre_by_id(genre_id)
        fetch_by_id(self.db, Book, book_id, "Book not found")
        ensure_association_does_not_exist(
            self.db, BookGenre, genre_id=genre_id, book_id=book_id
```

```
        )
        self.db.add(BookGenre(genre_id=genre_id, book_id=book_id))
        self.db.commit()

    def remove_genre_book_association(self, genre_id: int, book_id: int):
        self.get_genre_by_id(genre_id)
        fetch_by_id(self.db, Book, book_id, "Book not found")
        association = fetch_association(
            self.db,
            BookGenre,
            "Association not found",
            genre_id=genre_id,
            book_id=book_id,
        )
        self.db.delete(association)
        self.db.commit()
```

app/crud/api/v1/users.py

```
from sqlalchemy.orm import Session
from passlib.context import CryptContext
from fastapi import HTTPException
from app.models.user import User
from app.schemas.api.v1.user import SignUpSchema, SignInSchema, UpdateUserSchema
from app.crud.shared.db_utils import ensure_unique, fetch_by_attr


class UsersCrud:
    def __init__(self, db: Session):
        self.db = db
        self.pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

    def sign_in_user(self, user_data: SignInSchema):
        user = fetch_by_attr(self.db, User, "email", user_data.email, "User not
found")
        if not self._verify_password(user_data.password, user.hashed_password):
            raise HTTPException(status_code=401, detail="Invalid password")
        return user

    def sign_up_user(self, user_data: SignUpSchema):
        ensure_unique(self.db, User, "email", user_data.email, "Email already in
use")
```

```
        user_params = user_data.model_dump()
        user_params["hashed_password"] = self._get_password_hash(
            user_params.pop("password")
        )
        user = User(**user_params)
        self.db.add(user)
        self.db.commit()
        self.db.refresh(user)
        return user

    def update_user(self, user: User, user_data: UpdateUserSchema):
        user_params = user_data.model_dump(exclude_unset=True)
        if "email" in user_params and user_data.email != user.email:
            ensure_unique(
                self.db, User, "email", user_data.email, "Email already in use"
            )
        if "password" in user_params:
            user_params["hashed_password"] = self._get_password_hash(
                user_params.pop("password")
            )
        for key, value in user_params.items():
            setattr(user, key, value)
        self.db.commit()
        self.db.refresh(user)
        return user

    def remove_user(self, user: User):
        self.db.delete(user)
        self.db.commit()

    def _get_password_hash(self, password: str):
        return self.pwd_context.hash(password)

    def _verify_password(self, plain_password: str, hashed_password: str):
        return self.pwd_context.verify(plain_password, hashed_password)
```

app/models/author.py

```
from sqlalchemy import Column, Integer, String, Text, DateTime
from sqlalchemy.sql import func
from sqlalchemy.orm import relationship
from app.config import Base
```

```
class Author(Base):
    __tablename__ = "authors"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String)
    surname = Column(String)
    year_of_birth = Column(Integer)
    biography = Column(Text)
    created_at = Column(DateTime, default=func.now())
    updated_at = Column(DateTime, default=func.now(), onupdate=func.now())

    books = relationship("Book", secondary="book_author",
back_populates="authors")
```

app/models/book.py

```
from sqlalchemy import Column, Integer, String, Text, DateTime
from sqlalchemy.sql import func
from sqlalchemy.orm import relationship
from app.config import Base


class Book(Base):
    __tablename__ = "books"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String)
    description = Column(Text)
    year_of_publication = Column(Integer)
    isbn = Column(String, unique=True)
    series = Column(String)
    file_link = Column(String)
    edition = Column(String)
    created_at = Column(DateTime, default=func.now())
    updated_at = Column(DateTime, default=func.now(), onupdate=func.now())

    authors = relationship("Author", secondary="book_author",
back_populates="books")
    genres = relationship("Genre", secondary="book_genre", back_populates="books")
```

app/models/book_author.py

```
from sqlalchemy import Column, Integer, ForeignKey, DateTime, func
from app.config import Base


class BookAuthor(Base):
    __tablename__ = "book_author"

    book_id = Column(
        Integer, ForeignKey("books.id", ondelete="CASCADE"), primary_key=True
    )
    author_id = Column(
        Integer, ForeignKey("authors.id", ondelete="CASCADE"), primary_key=True
    )
    created_at = Column(DateTime, default=func.now())
```

app/models/book_genre.py

```
from sqlalchemy import Column, Integer, ForeignKey, DateTime, func
from app.config import Base


class BookGenre(Base):
    __tablename__ = "book_genre"

    book_id = Column(
        Integer, ForeignKey("books.id", ondelete="CASCADE"), primary_key=True
    )
    genre_id = Column(
        Integer, ForeignKey("genres.id", ondelete="CASCADE"), primary_key=True
    )
    created_at = Column(DateTime, default=func.now())
```

app/models/genre.py

```
from sqlalchemy import Column, Integer, String, Text, DateTime
from sqlalchemy.sql import func
from sqlalchemy.orm import relationship
from app.config import Base


class Genre(Base):
```

```
    __tablename__ = "genres"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String)
    description = Column(Text)
    created_at = Column(DateTime, default=func.now())
    updated_at = Column(DateTime, default=func.now(), onupdate=func.now())
    books = relationship("Book", secondary="book_genre", back_populates="genres")
```

app/models/user.py

```
from sqlalchemy import Column, Integer, String, DateTime
from sqlalchemy.sql import func
from app.config import Base


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    name = Column(String)
    surname = Column(String)
    avatar_link = Column(String)
    access_level = Column(Integer, default=0)
    created_at = Column(DateTime, default=func.now())
    updated_at = Column(DateTime, default=func.now(), onupdate=func.now())

    def is_user(self):
        return self.access_level == 0

    def is_librarian(self):
        return self.access_level == 1

    def is_admin(self):
        return self.access_level == 2
```

app/routers/api/v1/authors.py

```
from fastapi import APIRouter, Depends, Response
from app.schemas.api.v1.author import (
    AuthorSchema,
```

```
    CreateAuthorSchema,
    UpdateAuthorSchema,
    AuthorSortingSchema,
    author_search_dependency,
)
from app.schemas.api.v1.book import (
    BookSchema,
    BookSortingSchema,
    book_search_dependency,
)
from app.schemas.pagination import PaginationParams, PaginatedResponse
from app.crud.api.v1.authors import AuthorsCrud
from app.routers.shared.response_templates import (
    not_found_response,
    bad_request_response,
    invalid_authentication_responses,
    filtering_validation_error_response,
    combine_responses,
)
from app.routers.api.v1.shared.depends import get_authors_crud, get_librarian_user

router = APIRouter()


@router.get(
    "/",
    response_model=PaginatedResponse[AuthorSchema],
    responses=filtering_validation_error_response(),
)
async def get_authors(
    filters: dict = Depends(author_search_dependency),
    sorting_params: AuthorSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: AuthorsCrud = Depends(get_authors_crud),
):
    return crud.get_authors(
        filters=filters, sorting_params=sorting_params, pagination=pagination
    )


@router.get(
    "/{author_id}", response_model=AuthorSchema,
```

```
responses=not_found_response("author")
)
async def get_author(author_id: int, crud: AuthorsCrud =
Depends(get_authors_crud)):
    return crud.get_author_by_id(author_id=author_id)


@router.post(
    "/",
    response_model=AuthorSchema,
    status_code=201,
    responses=invalid_authentication_responses(),
)
async def create_author(
    author: CreateAuthorSchema,
    crud: AuthorsCrud = Depends(get_authors_crud),
    current_user=Depends(get_librarian_user),
):
    return crud.create_author(author_data=author)


@router.patch(
    "/{author_id}",
    response_model=AuthorSchema,
    responses=combine_responses(
        not_found_response("author"), invalid_authentication_responses()
    ),
)
async def update_author(
    author_id: int,
    author: UpdateAuthorSchema,
    crud: AuthorsCrud = Depends(get_authors_crud),
    current_user=Depends(get_librarian_user),
):
    return crud.update_author(author_id=author_id, author_data=author)


@router.delete(
    "/{author_id}",
    status_code=204,
    responses=combine_responses(
        not_found_response("author"), invalid_authentication_responses()
```

```
    ),
)
async def delete_author(
    author_id: int,
    crud: AuthorsCrud = Depends(get_authors_crud),
    current_user=Depends(get_librarian_user),
):
    crud.remove_author(author_id=author_id)
    return Response(status_code=204)


@router.get(
    "/{author_id}/books",
    response_model=PaginatedResponse[BookSchema],
    responses=combine_responses(
        not_found_response("author"), filtering_validation_error_response()
    ),
)
def get_books_of_author(
    author_id: int,
    filters: dict = Depends(book_search_dependency),
    sorting_params: BookSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: AuthorsCrud = Depends(get_authors_crud),
):
    return crud.get_books_of_author(
        author_id=author_id,
        filters=filters,
        sorting_params=sorting_params,
        pagination=pagination,
    )


@router.post(
    "/{author_id}/books/{book_id}",
    status_code=201,
    responses=combine_responses(
        not_found_response("author"),
        not_found_response("book"),
        bad_request_response("Association already exists"),
        invalid_authentication_responses(),
    ),
```

```
)
def create_author_book_association(
    author_id: int,
    book_id: int,
    crud: AuthorsCrud = Depends(get_authors_crud),
    current_user=Depends(get_librarian_user),
):
    crud.create_author_book_association(author_id=author_id, book_id=book_id)
    return Response(status_code=201)


@router.delete(
    "/{author_id}/books/{book_id}",
    status_code=204,
    responses=combine_responses(
        not_found_response("author"),
        not_found_response("book"),
        not_found_response("association"),
        invalid_authentication_responses(),
    ),
)
async def delete_author_book_association(
    author_id: int,
    book_id: int,
    crud: AuthorsCrud = Depends(get_authors_crud),
    current_user=Depends(get_librarian_user),
):
    crud.remove_author_book_association(author_id=author_id, book_id=book_id)
    return Response(status_code=204)
```

app/routers/api/v1/books.py

```
from fastapi import APIRouter, Depends, Response
from app.schemas.api.v1.book import (
    BookSchema,
    CreateBookSchema,
    UpdateBookSchema,
    BookSortingSchema,
    book_search_dependency,
)
from app.schemas.api.v1.author import (
    AuthorSchema,
```

```
    AuthorSortingSchema,
    author_search_dependency,
)
from app.schemas.api.v1.genre import (
    GenreSchema,
    GenreSortingSchema,
    genre_search_dependency,
)
from app.schemas.pagination import PaginationParams, PaginatedResponse
from app.crud.api.v1.books import BooksCrud
from app.routers.shared.response_templates import (
    not_found_response,
    bad_request_response,
    invalid_authentication_responses,
    filtering_validation_error_response,
    combine_responses,
)
from app.routers.api.v1.shared.depends import get_books_crud, get_librarian_user

router = APIRouter()


@router.get(
    "/",
    response_model=PaginatedResponse[BookSchema],
    responses=filtering_validation_error_response(),
)
async def get_books(
    filters: dict = Depends(book_search_dependency),
    sorting_params: BookSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: BooksCrud = Depends(get_books_crud),
):
    return crud.get_books(
        filters=filters,
        sorting_params=sorting_params,
        pagination=pagination,
    )


@router.get(
    "/{book_id}", response_model=BookSchema, responses=not_found_response("book")
```

```
)
async def get_book(book_id: int, crud: BooksCrud = Depends(get_books_crud)):
    return crud.get_book_by_id(book_id=book_id)


@router.post(
    "/",
    response_model=BookSchema,
    status_code=201,
    responses=combine_responses(
        bad_request_response("ISBN must be unique"),
invalid_authentication_responses()
    ),
)
async def create_book(
    book: CreateBookSchema,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    return crud.create_book(book_data=book)


@router.patch(
    "/{book_id}",
    response_model=BookSchema,
    responses=combine_responses(
        bad_request_response("ISBN must be unique"),
        not_found_response("book"),
        invalid_authentication_responses(),
    ),
)
async def update_book(
    book_id: int,
    book: UpdateBookSchema,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    return crud.update_book(book_id=book_id, book_data=book)


@router.delete(
    "/{book_id}",
```

```
    status_code=204,
    responses=combine_responses(
        not_found_response("book"), invalid_authentication_responses()
    ),
)
async def delete_book(
    book_id: int,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    crud.remove_book(book_id=book_id)
    return Response(status_code=204)


@router.get(
    "/{book_id}/authors",
    response_model=PaginatedResponse[AuthorSchema],
    responses=combine_responses(
        not_found_response("book"), filtering_validation_error_response()
    ),
)
def get_authors_of_book(
    book_id: int,
    filters: dict = Depends(author_search_dependency),
    sorting_params: AuthorSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: BooksCrud = Depends(get_books_crud),
):
    return crud.get_authors_of_book(
        book_id=book_id,
        filters=filters,
        sorting_params=sorting_params,
        pagination=pagination,
    )


@router.post(
    "/{book_id}/authors/{author_id}",
    status_code=201,
    responses=combine_responses(
        not_found_response("book"),
        not_found_response("author"),
```

```
        bad_request_response("Association already exists"),
        invalid_authentication_responses(),
    ),
)
def create_book_author_association(
    book_id: int,
    author_id: int,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    crud.create_book_author_association(book_id=book_id, author_id=author_id)
    return Response(status_code=201)


@router.delete(
    "/{book_id}/authors/{author_id}",
    status_code=204,
    responses=combine_responses(
        not_found_response("book"),
        not_found_response("author"),
        not_found_response("association"),
        invalid_authentication_responses(),
    ),
)
async def delete_book_author_association(
    book_id: int,
    author_id: int,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    crud.remove_book_author_association(book_id=book_id, author_id=author_id)
    return Response(status_code=204)


@router.get(
    "/{book_id}/genres",
    response_model=PaginatedResponse[GenreSchema],
    responses=combine_responses(
        not_found_response("book"), filtering_validation_error_response()
    ),
)
def get_genres_of_book(
```

```
    book_id: int,
    filters: dict = Depends(genre_search_dependency),
    sorting_params: GenreSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: BooksCrud = Depends(get_books_crud),
):
    return crud.get_genres_of_book(
        book_id=book_id,
        filters=filters,
        sorting_params=sorting_params,
        pagination=pagination,
    )


@router.post(
    "/{book_id}/genres/{genre_id}",
    status_code=201,
    responses=combine_responses(
        not_found_response("book"),
        not_found_response("genre"),
        bad_request_response("Association already exists"),
        invalid_authentication_responses(),
    ),
)
def create_book_genre_association(
    book_id: int,
    genre_id: int,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    crud.create_book_genre_association(book_id=book_id, genre_id=genre_id)
    return Response(status_code=201)


@router.delete(
    "/{book_id}/genres/{genre_id}",
    status_code=204,
    responses=combine_responses(
        not_found_response("book"),
        not_found_response("genre"),
        not_found_response("association"),
        invalid_authentication_responses(),
```

```
    ),
)
async def delete_book_genre_association(
    book_id: int,
    genre_id: int,
    crud: BooksCrud = Depends(get_books_crud),
    current_user=Depends(get_librarian_user),
):
    crud.remove_book_genre_association(book_id=book_id, genre_id=genre_id)
    return Response(status_code=204)
```

app/routers/api/v1/genres.py

```
from fastapi import APIRouter, Depends, Response
from app.schemas.api.v1.genre import (
    GenreSchema,
    CreateGenreSchema,
    UpdateGenreSchema,
    GenreSortingSchema,
    genre_search_dependency,
)
from app.schemas.api.v1.book import (
    BookSchema,
    BookSortingSchema,
    book_search_dependency,
)
from app.schemas.pagination import PaginationParams, PaginatedResponse
from app.crud.api.v1.genres import GenresCrud
from app.routers.shared.response_templates import (
    not_found_response,
    bad_request_response,
    invalid_authentication_responses,
    filtering_validation_error_response,
    combine_responses,
)
from app.routers.api.v1.shared.depends import get_genres_crud, get_librarian_user

router = APIRouter()


@router.get(
    "/",
```

```
    response_model=PaginatedResponse[GenreSchema],
    responses=filtering_validation_error_response(),
)
async def get_genres(
    filters: dict = Depends(genre_search_dependency),
    sorting_params: GenreSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: GenresCrud = Depends(get_genres_crud),
):
    return crud.get_genres(
        filters=filters, sorting_params=sorting_params, pagination=pagination
    )


@router.get(
    "/{genre_id}", response_model=GenreSchema,
responses=not_found_response("genre")
)
async def get_genre(genre_id: int, crud: GenresCrud = Depends(get_genres_crud)):
    return crud.get_genre_by_id(genre_id=genre_id)


@router.post(
    "/",
    response_model=GenreSchema,
    status_code=201,
    responses=invalid_authentication_responses(),
)
async def create_genre(
    genre: CreateGenreSchema,
    crud: GenresCrud = Depends(get_genres_crud),
    current_user=Depends(get_librarian_user),
):
    return crud.create_genre(genre_data=genre)


@router.patch(
    "/{genre_id}",
    response_model=GenreSchema,
    responses=combine_responses(
        not_found_response("genre"), invalid_authentication_responses()
    ),
```

```
)
async def update_genre(
    genre_id: int,
    genre: UpdateGenreSchema,
    crud: GenresCrud = Depends(get_genres_crud),
    current_user=Depends(get_librarian_user),
):
    return crud.update_genre(genre_id=genre_id, genre_data=genre)


@router.delete(
    "/{genre_id}",
    status_code=204,
    responses=combine_responses(
        not_found_response("genre"), invalid_authentication_responses()
    ),
)
async def delete_genre(
    genre_id: int,
    crud: GenresCrud = Depends(get_genres_crud),
    current_user=Depends(get_librarian_user),
):
    crud.remove_genre(genre_id=genre_id)
    return Response(status_code=204)


@router.get(
    "/{genre_id}/books",
    response_model=PaginatedResponse[BookSchema],
    responses=combine_responses(
        not_found_response("genre"), filtering_validation_error_response()
    ),
)
def get_books_of_genre(
    genre_id: int,
    filters: dict = Depends(book_search_dependency),
    sorting_params: BookSortingSchema = Depends(),
    pagination: PaginationParams = Depends(),
    crud: GenresCrud = Depends(get_genres_crud),
):
    return crud.get_books_of_genre(
        genre_id=genre_id,
```

```
        filters=filters,
        sorting_params=sorting_params,
        pagination=pagination,
    )


@router.post(
    "/{genre_id}/books/{book_id}",
    status_code=201,
    responses=combine_responses(
        not_found_response("genre"),
        not_found_response("book"),
        bad_request_response("Association already exists"),
        invalid_authentication_responses(),
    ),
)
def create_genre_book_association(
    genre_id: int,
    book_id: int,
    crud: GenresCrud = Depends(get_genres_crud),
    current_user=Depends(get_librarian_user),
):
    crud.create_genre_book_association(genre_id=genre_id, book_id=book_id)
    return Response(status_code=201)


@router.delete(
    "/{genre_id}/books/{book_id}",
    status_code=204,
    responses=combine_responses(
        not_found_response("genre"),
        not_found_response("book"),
        not_found_response("association"),
        invalid_authentication_responses(),
    ),
)
async def delete_genre_book_association(
    genre_id: int,
    book_id: int,
    crud: GenresCrud = Depends(get_genres_crud),
    current_user=Depends(get_librarian_user),
):
```

```
    crud.remove_genre_book_association(genre_id=genre_id, book_id=book_id)
    return Response(status_code=204)
```

app/routers/api/v1/sessions.py

```
from fastapi import APIRouter, Depends, Response
from app.crud.api.v1.users import UsersCrud
from app.schemas.api.v1.user import (
    UserSchema,
    SignUpSchema,
    SignInSchema,
    UpdateUserSchema,
)
from app.routers.api.v1.shared.depends import get_users_crud
from app.services.authorization import create_jwt_token, get_current_user
from app.schemas.token import Token
from app.routers.shared.response_templates import (
    bad_request_response,
    invalid_authentication_responses,
    invalid_password_response,
    not_found_response,
    combine_responses,
)

router = APIRouter()


@router.post(
    "/sign_up", status_code=201, responses=bad_request_response("Email already in
use")
)
async def sign_up(user_data: SignUpSchema, crud: UsersCrud =
Depends(get_users_crud)):
    crud.sign_up_user(user_data)
    return Response(status_code=201)


@router.post(
    "/sign_in",
    response_model=Token,
    responses=combine_responses(
        invalid_password_response(),
```

```
        not_found_response("User"),
    ),
)
async def sign_in(
    user_data: SignInSchema,
    crud: UsersCrud = Depends(get_users_crud),
):
    user = crud.sign_in_user(user_data)
    return create_jwt_token(user.id)


@router.get(
    "/current_user",
    response_model=UserSchema,
    responses=invalid_authentication_responses(),
)
async def show_current_user(current_user=Depends(get_current_user)):
    return current_user


@router.patch(
    "/current_user",
    response_model=UserSchema,
    responses=combine_responses(
        invalid_authentication_responses(),
        bad_request_response("Email already in use"),
    ),
)
async def update_current_user(
    user_data: UpdateUserSchema,
    current_user=Depends(get_current_user),
    crud: UsersCrud = Depends(get_users_crud),
):
    return crud.update_user(current_user, user_data)


@router.delete(
    "/current_user", status_code=204, responses=invalid_authentication_responses()
)
async def delete_current_user(
    current_user=Depends(get_current_user),
    crud: UsersCrud = Depends(get_users_crud),
```

```
):
    crud.remove_user(current_user)
    return Response(status_code=204)
```

app/schemas/api/v1/author.py

```
from datetime import datetime
from typing import Literal
from pydantic import BaseModel, Field, ConfigDict
from fastapi import Query
from fastapi.openapi.models import Example

YearField = Field(..., ge=1000, le=9999)


class AuthorSchema(BaseModel):
    id: int
    name: str
    surname: str
    year_of_birth: int = YearField
    biography: str
    created_at: datetime
    updated_at: datetime


class CreateAuthorSchema(BaseModel):
    name: str
    surname: str
    year_of_birth: int = YearField
    biography: str | None = ""

    model_config = ConfigDict(extra="forbid")


class UpdateAuthorSchema(BaseModel):
    name: str | None = None
    surname: str | None = None
    year_of_birth: int | None = Field(None, ge=1000, le=9999)
    biography: str | None = None

    model_config = ConfigDict(extra="forbid")
```

```
class AuthorSortingSchema(BaseModel):
    sort_by: (
        Literal[
            "name",
            "surname",
            "year_of_birth",
            "biography",
            "created_at",
            "updated_at",
        ]
        | None
    ) = Query(None)
    sort_order: Literal["asc", "desc"] | None = Query(None)


class AuthorSearchSchema(BaseModel):
    name: str | None = None
    surname: str | None = None
    year_of_birth: str | None = None
    biography: str | None = None
    created_at: str | None = None
    updated_at: str | None = None


def author_search_dependency(
    name: str | None = Query(
        default=None,
        description="Author's name; supports operators: eq (default), ne, like,
ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Case-insensitive name search",
                value="ilike:%john%",
            ),
            "example2": Example(
                summary="Exact name match",
                value="eq:John",
            ),
            "example3": Example(
                summary="Multiple names",
                value="in:John,Jane,Mark",
```

```
            ),
        },
    ),
    surname: str | None = Query(
        default=None,
        description="Author's surname; supports operators: eq (default), ne, like,
ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Partial surname match",
                value="like:%doe%",
            ),
            "example2": Example(
                summary="Case-insensitive surname search",
                value="ilike:%smith%",
            ),
        },
    ),
    year_of_birth: str | None = Query(
        default=None,
        description="Year of birth; supports operators: eq (default), ne, lt, lte,
gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="Born after 1950",
                value="gte:1950",
            ),
            "example2": Example(
                summary="Exact birth year",
                value="eq:1975",
            ),
            "example3": Example(
                summary="Multiple birth years",
                value="in:1950,1960,1970",
            ),
        },
    ),
    biography: str | None = Query(
        default=None,
        description="Biography; supports operators: eq (default), ne, like, ilike,
in",
        openapi_examples={
```

```
            "example1": Example(
                summary="Mention of 'Nobel'",
                value="ilike:%Nobel%",
            ),
            "example2": Example(
                summary="Partial biography content",
                value="like:%poet%",
            ),
        },
    ),
    created_at: str | None = Query(
        default=None,
        description="Creation timestamp; supports operators: eq (default), ne, lt,
lte, gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="Created before 2023",
                value="lt:2023-01-01T00:00:00",
            ),
            "example2": Example(
                summary="Exact creation timestamp",
                value="eq:2022-06-01T12:00:00",
            ),
        },
    ),
    updated_at: str | None = Query(
        default=None,
        description="Update timestamp; supports operators: eq (default), ne, lt,
lte, gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="Updated after 2022",
                value="gte:2022-01-01T00:00:00",
            ),
            "example2": Example(
                summary="Exclude specific update timestamp",
                value="ne:2023-07-01T00:00:00",
            ),
        },
    ),
) -> AuthorSearchSchema:
    return AuthorSearchSchema(
```

```
        name=name,
        surname=surname,
        year_of_birth=year_of_birth,
        biography=biography,
        created_at=created_at,
        updated_at=updated_at,
    ).model_dump(exclude_none=True)
```

app/schemas/api/v1/book.py

```
from datetime import datetime
from typing import Literal
from pydantic import BaseModel, Field, ConfigDict
from fastapi import Query
from fastapi.openapi.models import Example

YearField = Field(..., ge=1000, le=9999)
ISBNField = Field(..., min_length=13, max_length=13, pattern=r"^\d{13}$")


class BookSchema(BaseModel):
    id: int
    title: str
    description: str
    year_of_publication: int = YearField
    isbn: str = ISBNField
    series: str
    file_link: str
    edition: str
    created_at: datetime
    updated_at: datetime


class CreateBookSchema(BaseModel):
    title: str
    description: str | None = ""
    year_of_publication: int = YearField
    isbn: str = ISBNField
    series: str | None = ""
    file_link: str | None = ""
    edition: str | None = ""
```

```
    model_config = ConfigDict(extra="forbid")


class UpdateBookSchema(BaseModel):
    title: str | None = None
    description: str | None = None
    year_of_publication: int | None = Field(None, ge=1000, le=9999)
    isbn: str | None = Field(None, min_length=13, max_length=13,
pattern=r"^\d{13}$")
    series: str | None = None
    file_link: str | None = None
    edition: str | None = None

    model_config = ConfigDict(extra="forbid")


class BookSortingSchema(BaseModel):
    sort_by: (
        Literal[
            "title",
            "description",
            "year_of_publication",
            "isbn",
            "series",
            "file_link",
            "edition",
            "created_at",
            "updated_at",
        ]
        | None
    ) = Query(None)
    sort_order: Literal["asc", "desc"] | None = Query(None)


class BookSearchSchema(BaseModel):
    title: str | None = None
    description: str | None = None
    year_of_publication: str | None = None
    isbn: str | None = None
    series: str | None = None
    file_link: str | None = None
    edition: str | None = None
```

```
    created_at: str | None = None
    updated_at: str | None = None


def book_search_dependency(
    title: str | None = Query(
        default=None,
        description="Book title; supports operators: eq (default), ne, lt, lte,
gt, gte, like, ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Case-insensitive match",
                value="ilike:%python%",
            ),
            "example2": Example(
                summary="Exact title match",
                value="eq:Python 101",
            ),
            "example3": Example(
                summary="Title list",
                value="in:Python 101,Python Tricks",
            ),
        },
    ),
    description: str | None = Query(
        default=None,
        description="Book description; supports operators: eq (default), ne, like,
ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Partial match on description",
                value="like:%advanced%",
            ),
            "example2": Example(
                summary="Case-insensitive search",
                value="ilike:%beginner%",
            ),
        },
    ),
    year_of_publication: str | None = Query(
        default=None,
        description="Year of publication; supports operators: eq (default), ne,
```

```
lt, lte, gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="Published after 2010",
                value="gte:2010",
            ),
            "example2": Example(
                summary="Exact year match",
                value="eq:2015",
            ),
            "example3": Example(
                summary="Year range",
                value="in:2010,2015,2020",
            ),
        },
    ),
    isbn: str | None = Query(
        default=None,
        description="ISBN number; supports operators: eq (default), ne, like,
ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Exact ISBN",
                value="eq:9783161484100",
            ),
            "example2": Example(
                summary="Partial ISBN match",
                value="like:%1484100",
            ),
        },
    ),
    series: str | None = Query(
        default=None,
        description="Book series; supports operators: eq (default), ne, like,
ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Exact series match",
                value="eq:Harry Potter",
            ),
            "example2": Example(
                summary="Case-insensitive search",
```

```
                value="ilike:%rings%",
            ),
        },
    ),
    file_link: str | None = Query(
        default=None,
        description="File link; supports operators: eq (default), ne, like, ilike,
in",
        openapi_examples={
            "example1": Example(
                summary="Partial URL match",
                value="like:%google.com%",
            ),
            "example2": Example(
                summary="Case-insensitive match",
                value="ilike:%dropbox%",
            ),
        },
    ),
    edition: str | None = Query(
        default=None,
        description="Edition; supports operators: eq (default), ne, like, ilike,
in",
        openapi_examples={
            "example1": Example(
                summary="Exact edition match",
                value="eq:2nd",
            ),
            "example2": Example(
                summary="Edition search",
                value="ilike:%revised%",
            ),
        },
    ),
    created_at: str | None = Query(
        default=None,
        description="Creation timestamp; supports operators: eq (default), ne, lt,
lte, gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="Before a specific date",
                value="lt:2023-01-01T00:00:00",
```

```
            ),
            "example2": Example(
                summary="Exact creation date",
                value="eq:2022-06-15T00:00:00",
            ),
        },
    ),
    updated_at: str | None = Query(
        default=None,
        description="Update timestamp; supports operators: eq (default), ne, lt,
lte, gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="After a specific date",
                value="gte:2022-01-01T00:00:00",
            ),
            "example2": Example(
                summary="Not equal to update date",
                value="ne:2023-06-15T00:00:00",
            ),
        },
    ),
) -> BookSearchSchema:
    return BookSearchSchema(
        title=title,
        description=description,
        year_of_publication=year_of_publication,
        isbn=isbn,
        series=series,
        file_link=file_link,
        edition=edition,
        created_at=created_at,
        updated_at=updated_at,
    ).model_dump(exclude_none=True)
```

app/schemas/api/v1/genre.py

```
from datetime import datetime
from typing import Literal
from pydantic import BaseModel, ConfigDict
from fastapi import Query
from fastapi.openapi.models import Example
```

```
class GenreSchema(BaseModel):
    id: int
    name: str
    description: str
    created_at: datetime
    updated_at: datetime


class CreateGenreSchema(BaseModel):
    name: str
    description: str | None = ""

    model_config = ConfigDict(extra="forbid")


class UpdateGenreSchema(BaseModel):
    name: str | None = None
    description: str | None = None

    model_config = ConfigDict(extra="forbid")


class GenreSortingSchema(BaseModel):
    sort_by: (
        Literal[
            "name",
            "description",
            "created_at",
            "updated_at",
        ]
        | None
    ) = Query(None)
    sort_order: Literal["asc", "desc"] | None = Query(None)


class GenreSearchSchema(BaseModel):
    name: str | None = None
    description: str | None = None
    created_at: str | None = None
    updated_at: str | None = None
```

```
def genre_search_dependency(
    name: str | None = Query(
        default=None,
        description="Genre name; supports operators: eq (default), ne, like,
ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Case-insensitive match",
                value="ilike:%fantasy%",
            ),
            "example2": Example(
                summary="Exact match",
                value="eq:fantasy",
            ),
            "example3": Example(
                summary="Multiple values",
                value="in:fantasy,drama,romance",
            ),
        },
    ),
    description: str | None = Query(
        default=None,
        description="Genre description; supports operators: eq (default), ne,
like, ilike, in",
        openapi_examples={
            "example1": Example(
                summary="Partial match",
                value="like:%myth%",
            ),
            "example2": Example(
                summary="Case-insensitive description",
                value="ilike:%legend%",
            ),
        },
    ),
    created_at: str | None = Query(
        default=None,
        description="Creation timestamp; supports operators: eq (default), ne, lt,
lte, gt, gte, in",
        openapi_examples={
```

```
            "example1": Example(
                summary="Before a specific date",
                value="lt:2023-01-01T00:00:00",
            ),
            "example2": Example(
                summary="Exact timestamp match",
                value="eq:2022-12-25T00:00:00",
            ),
            "example3": Example(
                summary="Between a range of dates",
                value="in:2022-01-01T00:00:00,2022-12-31T23:59:59",
            ),
        },
    ),
    updated_at: str | None = Query(
        default=None,
        description="Update timestamp; supports operators: eq (default), ne, lt,
lte, gt, gte, in",
        openapi_examples={
            "example1": Example(
                summary="After a specific date",
                value="gte:2022-01-01T00:00:00",
            ),
            "example2": Example(
                summary="Not equal to a specific update date",
                value="ne:2023-06-15T00:00:00",
            ),
        },
    ),
) -> GenreSearchSchema:
    return GenreSearchSchema(
        name=name,
        description=description,
        created_at=created_at,
        updated_at=updated_at,
    ).model_dump(exclude_none=True)
```

app/schemas/api/v1/user.py

```
import re
from pydantic import BaseModel, EmailStr, ConfigDict, field_validator
from datetime import datetime
```

```
PasswordPattern = r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]*$"


def validate_password(value: str) -> str:
    if len(value) < 8:
        raise ValueError("Password must be at least 8 characters long")
    if len(value) > 100:
        raise ValueError("Password must be at most 100 characters long")
    if not re.match(PasswordPattern, value):
        raise ValueError(
            "Password must contain at least one uppercase letter, one lowercase
letter and one digit"
        )
    return value


class UserSchema(BaseModel):
    id: int
    email: EmailStr
    name: str
    surname: str
    avatar_link: str
    access_level: int
    created_at: datetime
    updated_at: datetime


class SignUpSchema(BaseModel):
    name: str
    surname: str
    email: EmailStr
    password: str
    avatar_link: str

    model_config = ConfigDict(extra="forbid")

    @field_validator("password")
    def check_password(cls, value):
        return validate_password(value)
```

```
class UpdateUserSchema(BaseModel):
    name: str | None = None
    surname: str | None = None
    email: EmailStr | None = None
    password: str | None = None
    avatar_link: str | None = None

    model_config = ConfigDict(extra="forbid")

    @field_validator("password")
    def check_password(cls, value):
        if value is None:
            return value
        return validate_password(value)


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

    model_config = ConfigDict(extra="forbid")

    @field_validator("password")
    def check_password(cls, value):
        return validate_password(value)
```

app/schemas/pagination.py

```
from typing import TypeVar, Generic, Sequence
from pydantic import BaseModel
from fastapi import Query

T = TypeVar("T")


class PaginationParams(BaseModel):
    page: int = Query(
        1, gt=0, description="Page number, default is 1, must be greater than 0"
    )
    size: int = Query(
        50, gt=0, description="Page size, default is 50, must be greater than 0"
    )
```

```
class PaginatedResponse(BaseModel, Generic[T]):
    items: Sequence[T]
    total: int
    page: int
    size: int
    pages: int
```

app/schemas/token.py

```
from pydantic import BaseModel


class Token(BaseModel):
    access_token: str
    token_type: str
```

app/services/ authorization.py

```
import jwt
from enum import IntEnum
from datetime import datetime, timedelta, timezone
from fastapi import HTTPException, Depends
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from sqlalchemy import select
from app.config import settings
from app.schemas.token import Token
from app.config import get_db
from app.models.user import User

auth_bearer = HTTPBearer()


class Role(IntEnum):
    USER = 0
    LIBRARIAN = 1
    ADMIN = 2


def create_jwt_token(user_id: int):
    payload = {
```

```
        "sub": str(user_id),
        "exp": datetime.now(timezone.utc)
        + timedelta(minutes=settings.JWT_TOKEN_EXPIRATION),
    }
    token = jwt.encode(payload, settings.JWT_SECRET_KEY, algorithm="HS256")
    return Token(access_token=token, token_type="Bearer")


def decode_jwt_token(token: str):
    try:
        payload = jwt.decode(token, settings.JWT_SECRET_KEY, algorithms=["HS256"])
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token has expired")
    except jwt.PyJWTError:
        raise HTTPException(status_code=401, detail="Invalid token")


def get_current_user(
    auth: HTTPAuthorizationCredentials = Depends(auth_bearer),
    db=Depends(get_db),
):
    token = auth.credentials
    user_id = int(decode_jwt_token(token)["sub"])
    user = db.execute(select(User).where(User.id == user_id)).scalar_one_or_none()
    if not user:
        raise HTTPException(
            status_code=401, detail="Token pointing to non-existent user"
        )
    return user


def get_current_user_with_minimum_role(required_role: Role):
    def dependency(
        user: User = Depends(get_current_user),
    ):
        if user.access_level < required_role:
            raise HTTPException(status_code=403, detail="Insufficient rights")
        return user

    return dependency
```

app/services/pagination.py

```
from sqlalchemy import select, func
from sqlalchemy.orm import Session
from app.schemas.pagination import PaginationParams, PaginatedResponse


def paginate(
    db: Session, stmt: select, pagination: PaginationParams
) -> PaginatedResponse:
    offset = (pagination.page - 1) * pagination.size
    limit = pagination.size

    paginated_stmt = stmt.offset(offset).limit(limit)
    results = db.execute(paginated_stmt).scalars().all()
    subquery = stmt.subquery()
    total_count_stmt = select(func.count()).select_from(subquery)
    total_count = db.execute(total_count_stmt).scalar()
    total_pages = (total_count + pagination.size - 1) // pagination.size

    return PaginatedResponse(
        items=results,
        total=total_count,
        page=pagination.page,
        size=pagination.size,
        pages=total_pages,
    )
```

app/services/search.py

```
from fastapi import HTTPException
from sqlalchemy import select

ALLOWED_OPERATORS = {"eq", "ne", "lt", "lte", "gt", "gte", "like", "ilike", "in"}


def parse_filter(raw_value: str):
    if ":" in raw_value:
        operator, value = raw_value.split(":", 1)
        if operator not in ALLOWED_OPERATORS:
            raise HTTPException(
```

```
                status_code=422, detail=f"Unsupported filter operator:
'{operator}'"
            )
    else:
        operator = "eq"
        value = raw_value
    return operator, value


def apply_filters(stmt: select, filters: dict, allowed_fields: dict) -> select:
    operator_map = {
        "eq": lambda col, val: col == val,
        "ne": lambda col, val: col != val,
        "lt": lambda col, val: col < val,
        "lte": lambda col, val: col <= val,
        "gt": lambda col, val: col > val,
        "gte": lambda col, val: col >= val,
        "like": lambda col, val: col.like(val),
        "ilike": lambda col, val: col.ilike(val),
        "in": lambda col, val: col.in_(val.split(",")),
    }

    for field, raw_value in filters.items():
        if field not in allowed_fields:
            raise HTTPException(
                status_code=422, detail=f"Filtering by '{field}' is not allowed."
            )

        operator, value = parse_filter(raw_value)
        column = allowed_fields[field]

        stmt = stmt.where(operator_map[operator](column, value))

    return stmt
```

app/services/sorting.py

```
from sqlalchemy import asc, desc, select


def apply_sorting(stmt: select, sorting_params, sort_fields: dict) -> select:
    field = sorting_params.sort_by
```

```
order = sorting_params.sort_order
column = sort_fields[field]

return stmt.order_by(asc(column) if order == "asc" else desc(column))
```