**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

**До захисту допущено:**
**Завідувач кафедри**

_____ _____
(підпис)

“__” _____2025 р.

# Дипломний проєкт

**на здобуття ступеня бакалавра**

**за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”**

**спеціальності 123 “Інженерія програмного забезпечення”**

на тему: Веб-застосунок для рекомендацій книг на основі користувацьких уподобань

Виконав : студент __4__ курсу, групи ІО-16
(шифр групи)

Гадієв Ренат Анатолійович
(прізвище, ім’я, по батькові) (підпис)

Керівник ас. Валько В.В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ас. Пономаренко А.М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент к.т.н. ст. викл. каф. ІСТ Орленко Сергій Петрович
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 123 “Інженерія програмного забезпечення”

**ЗАТВЕРДЖУЮ**
**В.о. завідувача кафедри**
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

**ЗАВДАННЯ**
на бакалаврський дипломний проєкт студента

Гадієва Рената Анатолійовича

1. Тема проєкту *Веб-застосунок для рекомендацій книг на основі користувацьких уподобань*
керівник проєкту _____ас. Валько В.В._____
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від

2. Термін здачі студентом закінченого проєкту *2 червня 2025 року*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області

Розділ 2. Огляд технологій розробки.

Розділ 3. Реалізація застосунку.

Розділ 4. Огляд та тестування веб-застосунку.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| Нормоконтроль | Пономаренко А.М. | | |
| | | | |

7. Дата видачі завдання «6» січня 2025 р.

Календарний план

| № п/п | Найменування етапів дипломного проєкту | Терміни виконання етапів проєкту | Примітки |
|---|---|---|---|
| 1. | *Затвердження теми проєкту* | *23.12.2024 – 27.12.2024* | |
| *2.* | *Вивчення та аналіз завдання* | *27.12.2024 – 27.02.2025* | |
| *3.* | *Розробка архітектури та загальної структури системи* | *27.02.2025 – 28.03.2025* | |
| *4.* | *Розробка структур окремих підсистем* | *28.03.2025 – 28.04.2025* | |
| *5.* | *Програмна реалізація системи* | *28.04.2025 – 14.05.2025* | |
| 6. | *Оформлення пояснювальної записки* | *14.05.2025 – 20.05.2025* | |
| 7. | *Захист програмного продукту* | *10.06.2025* | |
| *8.* | *Передзахист* | *13.06.2025* | |
| *9.* | *Захист* | *21.06.2025* | |

Студент-дипломник _____ Ренат ГАДІЄВ
(підпис)

Керівник проєкту _____ Володимир ВАЛЬКО
(підпис)

## АНОТАЦІЯ

Дипломний проєкт присвячений розробці персоналізованої веб-системи рекомендацій книг на основі вподобань користувача. Основною метою є створення сервісу, який забезпечує точні та релевантні рекомендації, використовуючи контентно-орієнтовану рекомендаційну систему. Джерелом даних є публічне API бібліотеки OpenLibrary, що дозволяє отримувати інформацію про книги та авторів. Для обробки запитів використовується FastAPI, а механізми фільтрації реалізовані за допомогою TF-IDF-моделі та обчислення косинусної схожості між описами творів. Веб-інтерфейс реалізовано із використанням HTML-шаблонів, скриптів мовою програмування JavaScript та CSS-стилів, що дає можливість реалізувати сучасний та інтуїтивно зрозумілий для користувача інтерфейс.



## ANNOTATION

The diploma project is dedicated to the development of a personalized web-based book recommendation system based on user preferences. The main goal is to create a service that provides accurate and relevant recommendations using a content-based recommendation system. The data source is the OpenLibrary public API, which allows retrieving information about books and authors. FastAPI is used for request processing, and filtering mechanisms are implemented using a TF-IDF model and cosine similarity calculations between book descriptions. The web interface is developed using HTML templates, JavaScript programming language scripts, and CSS styles, enabling a modern and user-friendly interface.

| справки | Формат | *Значення* | *Найменування* | Кіл. листів | № екземпля | *Додаток* |
|---|---|---|---|---|---|---|
| | | | Документація загальна | | | |
| | | | Знову розроблена | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.002 ТЗ*** | Веб-застосунок для | 4 | | |
| | | | рекомендацій книг на основі | | | |
| | | | уподобань користувача | | | |
| | | | Технічне завдання | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.003 ПЗ*** | Веб-застосунок для | 67 | | |
| | | | рекомендацій книг на основі | | | |
| | | | уподобань користувача | | | |
| | | | Пояснювальна записка | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.004 Д1*** | Веб-застосунок для | 1 | | |
| | | | рекомендацій книг на основі | | | |
| | | | уподобань користувача | | | |
| | | | Схема роботи | | | |
| | | | веб-застосунку | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.4672008.005 Д2*** | Веб-застосунок для | 1 | | |
| | | | рекомендацій книг на основі | | | |
| | | | уподобань користувача | | | |
| | | | Структурна схема | | | |
| | | | архітектури веб-застосунку | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.006 Д3*** | Веб-застосунок для | 1 | | |
| | | | рекомендацій книг на основі | | | |
| | | | уподобань користувача | | | |
| | | | Маршрутизація запитів у веб-застосунку | | | |
| | | | | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.007 Д4*** | Веб-застосунок для | 55 | | |
| | | | рекомендацій книг на основі | | | |
| | | | уподобань користувача | | | |
| | | | Лістинг програмного коду | | | |
| | | | | | | |
| | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| *Зм* | ***Лист*** | ***№ докум.*** | ***Підп*** | ***Дата*** | | |
| ***Розро*** | | Гадієв Р.А. | | | | |
| ***Перев*** | | . | | | | |
| | | | | | | |

# *ІАЛЦ.467200.001 ОА*

*Веб-застосунок для рекомендацій книг на основі користувацьких уподобань*

Опис альбому

| Літ. | Аркуш | Аркушів |
|---|---|---|
| | 1 | 1 |

***НТУУ “КПІ” ФІОТ***
***ІО-16***

# ТЕХНІЧНЕ ЗАВДАННЯ
# ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для рекомендацій книг на основі уподобань користувача*»

Київ – 2025

# ЗМІСТ



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.002 ТЗ | | |
| | | № докум. | Підпис | Дата | | | |
| Розробив | | Гадієв Р.А.. | | | **Веб-застосунок для рекомендацій книг на основі користувацьких уподобань** Технічне завдання | Літ. | Аркуш 1 / Аркушів 4 |
| Перевірив | | | | | | | |
| | | | | | | НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-16 | |
| Н. Контр. | | | | | | | |
| Затвердив | | | | | | | |

# 1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку персоналізованого веб-сервісу для рекомендацій книг на основі користувацьких уподобань, а також на підтримку, розширення та оптимізацію розробленого сервісу.

Областю застосування цієї системи є онлайн-платформи для читачів, електронні бібліотеки, книжкові сервіси та освітні ресурси, а також окремі читачі, що бажають знайти нові книги відповідно до власних уподобань.

# 2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

# 3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення веб-системи, яка надає персоналізовані рекомендації книг, орієнтуючись на вподобання користувача, жанрові інтереси та тематичну близькість літературних творів.

Призначення системи полягає в забезпеченні зручного інструменту для підбору книжок, що відповідають читацьким інтересам, з метою покращення користувацького досвіду, підвищення зацікавленості у читанні та спрощення навігації у великому масиві літературної інформації.

# 4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами інформації, що використовувалася під час розробки веб-сервісу є існуючі сервіси, офіційна документація Open Library API, Fast API, інтернет статті з прикладами реалізації та використання окремих функцій.

# 5. ТЕХНІЧНІ ВИМОГИ

## 5.1 Вимоги до розробленого продукту

Розроблений веб-застосунок має задовольняти наступні вимоги:

- Простий та зрозумілий інтерфейс
- Універсальна система рекомендацій, яка забезпечує добір книг за різними критеріями – тематично подібною книгою, схожим автором або обраними жанрами
- Високий рівень швидкодії

## 5.2 Вимоги до програмного забезпечення:

- ОС Windows, MacOS, Linux
- IDE PyCharm 2021 версії або новіше
- Браузер

## 5.3 Вимоги до апаратної частини:

- ЦП не менше ніж AMD Ryzen 7 3750H
- ROM не менше ніж 64 ГБ
- RAM не менше ніж 8 ГБ

# 6. ЕТАПИ РОЗРОБКИ

| Назва етапів виконання | Термін виконання |
|---|---|
| Затвердження теми роботи | |
| Вивчення та аналіз завдання | |
| Розробка архітектури та загальної структури системи | |
| Розробка структур окремих частин системи | |
| Програмна реалізація системи | |
| Виправлення помилок | |
| Оформлення пояснювальної записки | |

# ПОЯСНЮВАЛЬНА ЗАПИСКА
# ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для рекомендацій книг на основі користувацьких уподобань*»

Київ – 2025

**ЗМІСТ**

# ПЕРЕЛІК СКОРОЧЕНЬ

| | |
|---|---|
| API | (Application Programming Interface) Інтерфейс прикладного програмування |
| CORS | (Cross-Origin Resource Sharing) Механізм обміну ресурсами між різними джерелами |
| DOM | (Document Object Model) Об’єктна модель документа |
| JSON | (JavaScript Object Notation) Формат обміну структурованими даними |
| JS | (JavaScript) Мова сценаріїв для динамічної поведінки веб-сторінок |
| OLID | (OpenLibrary Identifier) Унікальний ідентифікатор книги в OpenLibrary |
| REST | (Representational State Transfer) Архітектурний стиль веб-сервісів |
| TF-IDF | (Term Frequency – Inverse Document Frequency) Метод вагового оцінювання слів у текстах |
| FastAPI | (Fast Python Web Framework) Високопродуктивний фреймворк для створення API |

# ВСТУП

У сучасному світі обсяг доступної інформації стрімко зростає та охоплює практично всі сфери людської діяльності — від музики, відеоігор і розважальних матеріалів до наукових досліджень і аналітичних статей. З одного боку, це відкриває широкі можливості для розвитку, самореалізації та відпочинку, а з іншого — створює нові виклики, пов’язані зі складністю пошуку саме тієї інформації, яка відповідатиме особистим інтересам користувача.

Для вирішення проблеми безкінечного вибору під час пошуку необхідної інформації у більшості сучасних сервісів застосовуються рекомендаційні системи, які підбирають інформаційні об’єкти відповідно до уподобань користувача, дій людей зі схожими уподобаннями, матеріалів, що вже сподобалися користувачу. Використання рекомендаційних систем дозволяє завжди отримувати підбірки свіжих матеріалів відповідно до власних інтересів під час використання сервісів, що забезпечує приємний користувацький досвід.

Для книжкових сервісів створення якісних рекомендацій є особливо важливим через різноманітність доступної літератури та складність ручного пошуку релевантних видань серед тисяч доступних позицій. Саме тому впровадження системи персоналізованих рекомендацій у веб-застосунок підвищує ефективність взаємодії з користувачем і сприяє зростанню зацікавленості у читанні.

Метою даної дипломної роботи є дослідження принципів побудови рекомендаційних систем, аналіз їх переваг, недоліків та особливостей реалізації, а також розробка веб-застосунку для рекомендації книг на основі уподобань користувача. Фінальний програмний продукт має на меті заохочення до читання через надання персоналізованих рекомендацій, а реалізація у форматі веб-застосунку забезпечує його доступність із будь-якого пристрою, що підтримує браузер.

# РОЗДІЛ 1
# АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1. Огляд рекомендаційних систем

У сучасному інформаційному середовищі користувачі мають доступ до великої кількості цифрових матеріалів: музики, фільмів, книг, новин та інших джерел. З одного боку, це відкриває нові можливості для навчання, дозвілля та саморозвитку, проте водночас виникає проблема — складність у виборі. Часто буває важко знайти саме ті матеріали, які дійсно відповідають інтересам і потребам конкретної людини. Саме тому зростає значення інструментів, які допомагають підбирати інформацію відповідно до користувацьких уподобань. До таких інструментів належать рекомендаційні системи.

Рекомендаційні системи — це програми, які аналізують інформацію про користувача або про самі об’єкти, щоб автоматично пропонувати варіанти, які можуть зацікавити саме його. [1] Вони активно використовуються у багатьох популярних сервісах, таких як YouTube, Netflix, Amazon, Spotify тощо. У таких системах кожен користувач бачить власні рекомендації, які формуються на основі його активності, інтересів або схожості з іншими користувачами, що дозволяє якісно виділяти сервіс на тлі конкурентів.

Перші спроби створення подібних систем почались ще у 1990-х роках. Спочатку це були досить прості рішення, які показували найпопулярніші об’єкти або вручну створені списки. Згодом підходи стали складнішими: почали використовувати математичні моделі, методи машинного навчання, обробку великих обсягів даних.

Існує кілька основних підходів до побудови рекомендаційних систем:

**Контентно-орієнтований підхід (content-based filtering):**

Принцип: рекомендації формуються на основі властивостей самого об’єкта (наприклад, жанру, ключових слів у описі книги) та вподобань користувача. [2]

**Переваги:**

- не залежить від інших користувачів;
- працює навіть при малій кількості даних;
- добре підходить для нових систем.

**Недоліки:**

- складно рекомендувати щось зовсім нове, відмінне від попереднього вибору;
- немає “ефекту здивування” — система рекомендує те, що схоже на вже відоме.

**Колаборативна фільтрація (collaborative filtering):**

**Принцип:** система аналізує поведінку інших користувачів і пропонує об’єкти, які сподобалися тим, хто має подібні інтереси. [3]

**Переваги:**

- дозволяє відкривати нові, несподівані варіанти;
- працює незалежно від описів об’єктів.

**Недоліки:**

- потрібна велика кількість користувачів і даних;
- не працює добре, якщо користувач новий або об’єктів ще мало (проблема “холодного старту”).

**Гібридний підхід (hybrid):**

**Принцип:** поєднання декількох підходів, найчастіше контентного та колаборативного. [4]

**Переваги:**

- компенсує недоліки кожного з окремих методів;
- гнучкість і вища точність рекомендацій.

**Недоліки:**

- складність реалізації;
- потреба в більшій кількості ресурсів.

**Підхід на основі знань (knowledge-based):**

**Принцип:** система враховує чітко задані вимоги користувача, наприклад: “коротка книга про подорожі з хорошими відгуками”. [5]

**Переваги:**

- дає точні результати відповідно до критеріїв;
- не потребує даних про інших користувачів.

**Недоліки:**

- менш гнучкий;
- обмеженість, якщо користувач не знає, що саме шукає.

Для реалізації системи рекомендацій книг для дипломного проєкту було обрано контентно-орієнтований підхід. Він не потребує великої кількості даних про інших користувачів, що особливо зручно на етапі тестування або коли система ще не має постійної аудиторії. Такий варіант дозволяє працювати навіть тоді, коли кількість відгуків, оцінок чи збережених списків ще дуже мала.

У сфері рекомендацій книг такі системи є важливими, адже вони допомагають швидко знайти потрібну літературу, познайомитися з новими авторами, зацікавитися незнайомими жанрами та взагалі зробити процес

читання простішим і зручнішим. Тому подібні рішення активно застосовуються не лише на комерційних сайтах, але й у бібліотеках або освітніх платформах.

Відповідно, з появою відкритих джерел даних (OpenLibrary, Google Books API, Goodreads) стало можливо отримувати інформацію про книги без необхідності створення власної бази. Це зручно, але є й недолік — час відповіді на запити може бути більшим, особливо при великому навантаженні. Тому деякі системи використовують комбінований підхід: зберігають найпопулярніші книги локально, а рідкісні отримують через API за потреби.

### 1.2. Взаємодія з API для отримання інформації

Сучасні рекомендаційні системи для книг неможливі без якісних джерел даних, які найчастіше надаються через публічні програмні інтерфейси (API). Ці інструменти відіграють ключову роль, оскільки дозволяють автоматично отримувати структуровану інформацію про книги, авторів, жанри та інші метадані, що є життєво необхідним для функціонування будь-якої рекомендаційної системи.

Більшість сучасних книжкових API працюють за REST архітектурою, де запити формулюються у вигляді спеціальних URL-адрес, а відповіді повертаються у зручних для обробки форматах, переважно JSON або XML. [6] Для рекомендаційних систем особливе значення має не лише доступ до базової інформації про твори, а й можливість отримувати додаткові дані - рецензії, рейтинги, обкладинки та детальну жанрову класифікацію.

Оскільки існує кілька популярних API для роботи з книжковими даними, важливо зрозуміти, за якими критеріями варто оцінювати їх придатність для конкретної системи рекомендацій. Перш ніж перейти до детального порівняння таких рішень як OpenLibrary, Google Books та Goodreads API, варто чітко визначити вимоги до ідеального книжкового API.

Ключовими аспектами при виборі є: повнота наданих даних (наявність усіх необхідних полів), обмеження на кількість запитів, швидкість відповіді

сервера та ліцензійні умови використання. Кожен з існуючих API має свої сильні та слабкі сторони щодо цих параметрів. Наприклад, деякі рішення можуть пропонувати більш детальну жанрову класифікацію, тоді як інші - краще впораються із забезпеченням високої швидкості відповіді.

З метою вибору найращого джерела інформації до проекта проведено порівння найпопулярніших API.

### 1.2.1 OpenLibrary API

OpenLibrary API є частиною глобального некомерційного проєкту Internet Archive. API реалізоване у вигляді REST-сервісу, що дозволяє надсилати HTTP-запити з параметрами (наприклад, назва книги, автор, ключові слова) та отримувати відповіді у форматі JSON. Підтримується пошук, деталізація книги, доступ до авторів, жанрових тем, ідентифікаторів (ISBN, OLID), а також списків споріднених творів. [7]

**Переваги:**

- Повністю відкритий доступ без потреби авторизації
- Простий у використанні формат запитів
- Великий обсяг безкоштовної інформації про книги, авторів, жанри
- Підтримка семантичних зв’язків між творами (через subjects)
- Висока стабільність роботи API
- Підходить для побудови контентно-орієнтованих рекомендацій

**Недоліки:**

- Іноді дані неповні або дубльовані (особливо в авторських записах)
- Відсутність рейтингової або повноцінної рецензійної системи

### 1.2.2 Google Books API

Google Books API надає доступ до даних Google Книг, підтримує повнотекстовий пошук, фільтрацію за мовою, датою публікації, наявністю фрагментів. Запити реалізуються у вигляді GET-запитів з API-ключем, який видається через Google Cloud Console. [8]

**Переваги:**

- Можливість повнотекстового пошуку
- Велика кількість книг у різних мовах
- Підтримка даних про видавництво, обсяг, доступність, ISBN

**Недоліки:**

- Обов’язкова авторизація через API-ключ
- Добове обмеження кількості запитів (1000 безкоштовно)
- Відсутня чітка класифікація за жанрами або тематиками
- Нерегулярна структура відповідей (різна повнота для різних книг)

### 1.2.3 Goodreads API

Goodreads API дозволяло працювати з книжковою базою, що включала рейтинги, рецензії, цитати. Запити здійснювалися у форматі XML, доступ надавався після реєстрації. Платформа належить Amazon. [9]

**Переваги:**

- Доступ до унікальної бази рецензій користувачів
- Розвинена соціальна складова: цитати, огляди, читацькі добірки

**Недоліки:**

- Офіційно API більше не підтримується для нових розробників
- Працює тільки у форматі XML (не сучасний стандарт)
- Часто нестабільна робота, обмеження доступу

### 1.2.4 BookBrainz API

BookBrainz API є частиною відкритої ініціативи Metabrainz Foundation (аналог MusicBrainz). API реалізоване через REST-запити, відповіді повертаються у форматі JSON. Підтримується отримання інформації про твори, авторів, переклади, видання тощо. [10]

**Переваги:**

- Відкрите, документоване, гнучке API
- Підтримка складних зв’язків між сутностями (твори – видання – автори)

**Недоліки:**

- Порівняно невелика база літератури
- Недостатньо даних для побудови рекомендацій за жанрами
- Мала кількість супровідних метаданих (обкладинки, описи тощо)

### 1.2.5 Обґрунтування вибору OpenLibrary API

Серед розглянутих API найбільш доцільним для реалізації контентно-орієнтованої рекомендаційної системи виявився OpenLibrary API. У процесі аналізу було визначено, що він має низку переваг, які істотно спрощують розробку прототипу системи. Зокрема, API є відкритим, не потребує ключа доступу або реєстрації, що дозволяє одразу розпочати інтеграцію без додаткових налаштувань. Крім того, API вирізняється стабільністю та надає достатню кількість структурованих даних про книги, зокрема: назву, авторів, опис, рік публікації, тематику (subjects), обкладинку тощо.

Вказані атрибути зробили можливим використання OpenLibrary API як джерела даних для алгоритмів, які базуються на текстовій схожості (TF-IDF) та жанровому фільтруванні. Доступність тематичних зв’язків і ключових слів дозволила формувати рекомендації без необхідності зберігати локальні копії великої кількості об’єктів.

Порівняно з іншими доступними рішеннями, такими як Google Books API чи Goodreads API, OpenLibrary виявився менш обмеженим у використанні. Зокрема, Google Books потребує ключа, має ліміти на кількість запитів і не забезпечує повноцінної жанрової класифікації. Goodreads API наразі обмежений у доступі для нових користувачів, а підтримка API поступово згортається. Що стосується менш відомих рішень, таких як BookBrainz API, то вони ще не забезпечують повноти й глибини даних, необхідної для реалізації ефективної системи рекомендацій.

У результаті аналізу встановлено, що OpenLibrary API забезпечує найбільш оптимальне поєднання таких характеристик, як відкритість, функціональність, простота інтеграції та обсяг доступної інформації. Саме ці фактори стали вирішальними при виборі даного API для реалізації рекомендаційної системи в межах дипломного проєкту.

### 1.3 Огляд існуючих сервісів

У процесі створення власної системи рекомендацій книг доцільним є вивчення існуючих сервісів, які вже реалізують подібні функціональні можливості. Аналіз таких сервісів дозволяє глибше зрозуміти потреби кінцевих користувачів, оцінити різні підходи до реалізації систем рекомендацій, а також врахувати типові труднощі, що можуть виникати при розробці подібних рішень. Крім того, вивчення прикладів існуючих платформ надає можливість визначити найбільш ефективні архітектурні та алгоритмічні рішення для подальшої імплементації.

У межах цього підрозділу було розглянуто три популярні книжкові платформи, а саме: Goodreads, The StoryGraph та Whichbook. Кожен із зазначених сервісів реалізує власний підхід до формування рекомендацій, а також має певні особливості у способах взаємодії з користувачем. Це робить їх доцільними прикладами для порівняльного аналізу в контексті побудови власної рекомендаційної системи.

### 1.3.1 Goodreads

Goodreads — це один із найвідоміших і найстаріших онлайн-сервісів для пошуку та обговорення книг, заснований у 2006 році, а з 2013 року — у власності компанії Amazon. Платформа дозволяє користувачам створювати особисті книжкові полиці (shelves), виставляти оцінки, писати рецензії, додавати цитати, брати участь у спільнотах та обмінюватися рекомендаціями з іншими читачами. [11]

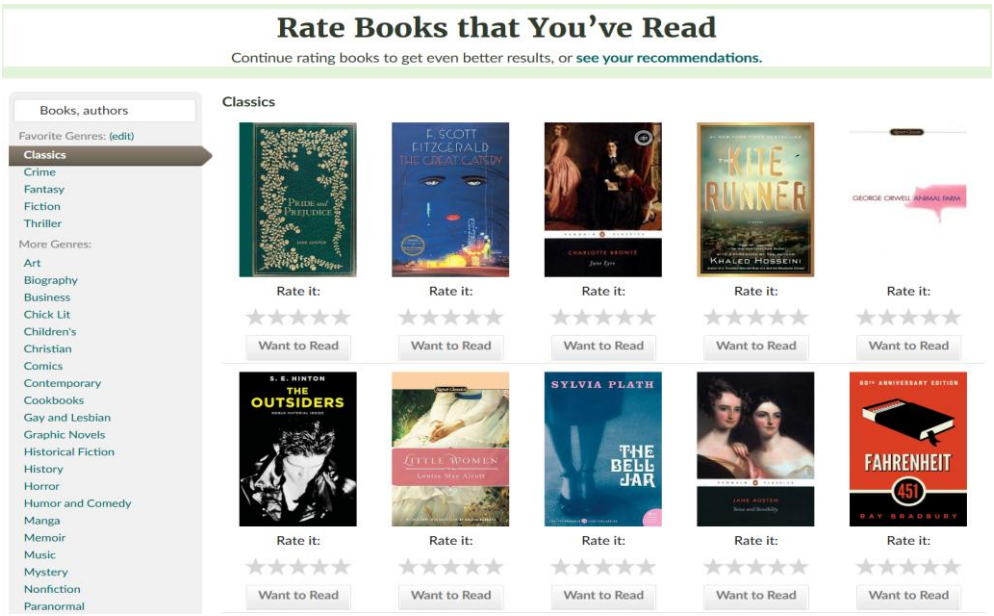



Рисунок 1.1 – Головна сторінка Goodreads

**Підхід до рекомендацій:**

Сервіс використовує колаборативну фільтрацію — рекомендації базуються на оцінках та поведінці користувача й інших читачів із подібними вподобаннями. Також враховуються дані про жанр, автора, популярність книги.

**Процес отримання рекомендацій:**

Після реєстрації користувач має змогу відмітити прочитані книги та поставити їм оцінки. На основі цих даних система автоматично формує розділ "Recommended for You", доступний у профілі. Також рекомендації

з’являються у вигляді списків на головній сторінці або в категорії “Because you liked…”. Користувач може взаємодіяти з цими книгами, додавати їх у список "Хочу прочитати" або переходити на сторінку твору для ознайомлення з рецензіями. Однак вплив користувача на налаштування логіки рекомендацій обмежений.

**Переваги:**

- Величезна база книг і рецензій
- Активна спільнота користувачів
- Соціальні функції: читальні групи, цитати, обговорення
- Можливість відстеження прогресу в читанні

**Недоліки:**

- Алгоритм рекомендацій є закритим і не завжди прозорим
- Обмежений контроль над налаштуваннями рекомендацій
- Інтерфейс дещо застарілий і перевантажений
- Обмежений розвиток API (із 2020 року API доступний лише для обмеженого кола партнерів)

### 1.3.2 The StoryGraph

The StoryGraph — сучасна платформа, орієнтована на глибоку персоналізацію книжкових рекомендацій. Сервіс надає багато аналітики щодо стилю книги, її емоційного тону, темпу, обсягу тощо. [12]

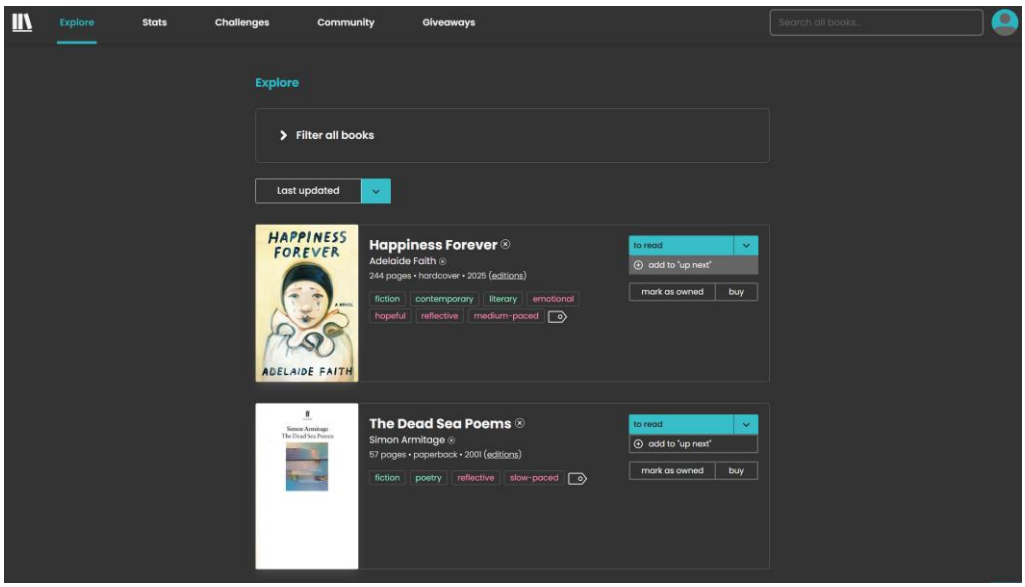


Рисунок 1.2 – Головна сторінка The StoryGraph

**Підхід до рекомендацій:**

Основу становить контентно-орієнтований підхід, зосереджений на детальних параметрах самих книг. Додатково враховується настрій, жанрова палітра, стиль викладу та читатський фідбек.

**Процес отримання рекомендацій:**

Користувач проходить коротке опитування про свої вподобання (у розділі “Personalized Recommendations”) або додає книги, які йому сподобались. На цій основі система формує динамічний список рекомендованих творів. Крім того, у розділі “Browse” доступний фільтр за параметрами: "повільна/динамічна", "світла/темна", "весела/сумна", кількість сторінок тощо.

Рекомендації оновлюються автоматично, а користувач може легко налаштовувати бажані параметри пошуку, що робить процес надзвичайно гнучким.

**Переваги:**

- Високий рівень персоналізації
- Сучасний і зручний інтерфейс

- Потужна статистика читання

**Недоліки:**

- Порівняно невелика спільнота користувачів
- Обмежений функціонал соціальних взаємодій
- Не всі книги містять повні метадані для глибокого аналізу

### 1.3.3 Whichbook

Whichbook — це нестандартна платформа для добору книг, яка ставить на перше місце емоційне сприйняття читання. Замість класичних фільтрів за жанром користувач самостійно визначає, якими має бути книга — легкою чи складною, веселою чи сумною, знайомою чи дивною. [13]

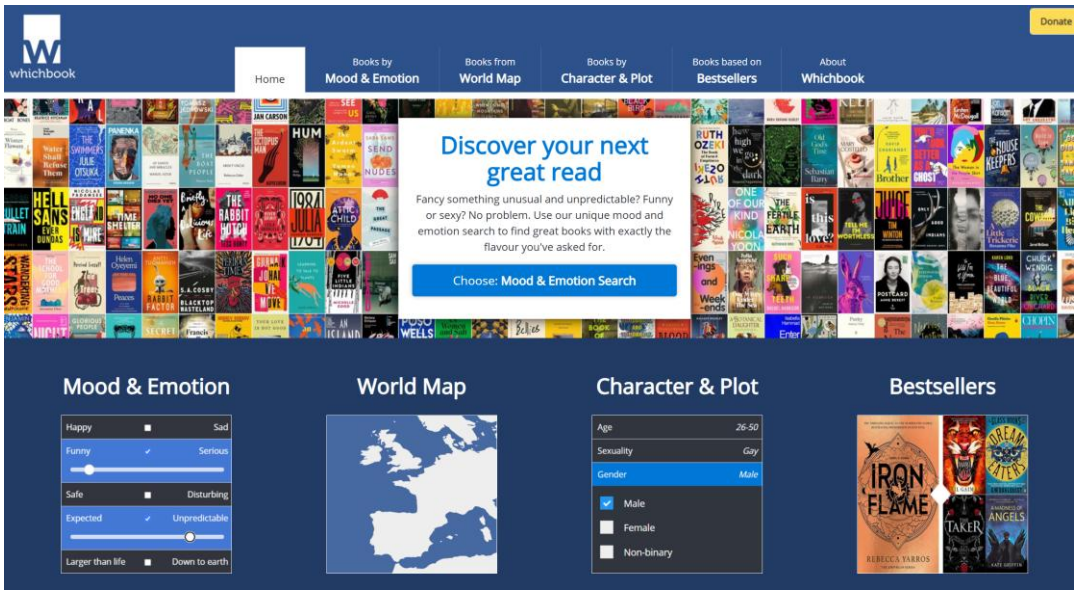



Рисунок 1.2 – Головна сторінка Whichbook

**Підхід до рекомендацій:**

Сервіс базується на ручній контентній класифікації: кожна книга оцінюється командою експертів за 12 параметрами, включаючи настрій, тематику, темп, стиль, культурний контекст тощо. Система не аналізує поведінку користувача, а підбирає книги лише за вибраними параметрами.

**Процес отримання рекомендацій:**

На головній сторінці користувач бачить повзунки (sliders), де може задати бажаний баланс — наприклад: “весела – сумна”, “оптимістична – песимістична”, “традиційна – експериментальна” тощо. Після налаштування декількох шкал, система одразу генерує список книг, які максимально відповідають емоційному профілю. Додатково можна фільтрувати за країною автора, мовою, наявністю аудіокниг, кількістю персонажів тощо. Усе це відбувається без реєстрації, і досвід максимально наближений до інтуїтивного вибору.

**Переваги:**

- Унікальний підхід до рекомендацій
- Гнучкий інтерфейс без потреби в акаунті
- Велика кількість емоційних та соціальних параметрів

**Недоліки:**

- Відсутність персоналізованих рекомендацій на основі історії читання
- Невелика база книжок порівняно з конкурентами
- Неможливість створити власну бібліотеку чи залишити рецензію

# ВИСНОВОК ДО РОЗДІЛУ 1

У цьому розділі було розглянуто основні підходи до реалізації рекомендаційних систем, існуючи джерела інформації, які потенційно можуть бути використані для реалізації програмного продукту за темою дипломного проекту, існуючі веб-сервіси для рекомендацій книг на основі користувацьких уподобань. Для основних підходів до створення рекомендаційних систем виокремлено переваги та недоліки кожного з підходів, проаналізовано можливість та доречність використання під час розробки власної системи. Розглянуто історію виникнення, принципи взаємодії, переваги та недоліки існуючих та доступних для використання джерел даних. Оглянуто історію, функціонал, та інтерфейс існуючих сервісів для рекомендацій книг.

На основі проведеного огляду та аналізу для розробки рекомендаційної системи у проєкті вирішено використати контентно-орієнтований підхід, головною перевагою якого є можливість створення якісних рекомендацій для користувача на основі конкретних матеріалів, без залежності від інших користувачів або історії переглядів, що гарно підходить для контексту дипломного проєкту і дозволяє уникнути проблеми “холодного старту”, яка виникає у разі використання інших підходів.

У якості джерела даних вирішено використовувати OpenLibrary API, яка на відміну від GoogleBooks API не має жорстких лімітів на запити протягом доби, дозволяє гнучко модифікувати запити для отримання інформації із використанням фільтрів, має великий обсяг доступних книг та авторів.

Огляд існуючих сервісів дав можливість порівняти можливості та користувацький інтерфейс, який кожен з них надає і сформувати вимоги до функціоналу та дизайну власного веб-застосунку.

# РОЗДІЛ 2. ОГЛЯД ТЕХНОЛОГІЙ РОЗРОБКИ

## 2.1 Мова програмування Python

Python — це високорівнева мова програмування загального призначення, яка була створена нідерландським розробником Ґвідо ван Россумом у 1989 році та вперше офіційно представлена в 1991 році. Основною метою розробки мови було створення інструменту, що дозволяє програмістам писати код, який є легким для читання і підтримки, при цьому залишаючись досить потужним для вирішення широкого кола задач. Python був натхненний мовами ABC, C, Modula-3 та іншими. З часом він набув популярності у науковому середовищі, сфері автоматизації, обробки даних та веб-розробці.

Існує кілька основних версій Python. Перехід від Python 2 до Python 3, який розпочався у 2008 році, став важливою віхою в розвитку мови, оскільки приніс кардинальні зміни в синтаксисі та внутрішній логіці, що дозволило усунути ряд проблем попередніх версій. Наразі підтримується лише гілка Python 3, яка постійно оновлюється та вдосконалюється. [14] На момент написання роботи актуальною є версія Python 3.12.

**Основні переваги Python:**

- Простота та читабельність синтаксису. Програмний код на Python є інтуїтивно зрозумілим і близьким до природної мови, що спрощує як написання, так і підтримку програм.
- Велика стандартна бібліотека. Python має широкий набір вбудованих модулів для роботи з текстом, файлами, мережею, процесами, структурованими даними та іншими компонентами.
- Наявність великої кількості сторонніх бібліотек і фреймворків. Завдяки відкритій спільноті доступні інструменти для веб-розробки (Django, FastAPI, Flask), машинного навчання (scikit-learn, TensorFlow),

обробки даних (Pandas, NumPy), автоматизації (Selenium, PyAutoGUI) та багато інших.

- Кросплатформеність. Python працює на більшості сучасних операційних систем, включаючи Windows, Linux, macOS, і дозволяє створювати переносні рішення.
- Розвинена інфраструктура тестування, розгортання та інтеграції. Мова підтримується великою кількістю інструментів для CI/CD, віртуального середовища (venv, conda), управління залежностями (pip, poetry) тощо.
- Широке застосування в різних галузях. Python активно використовується у веб-розробці, науці про дані, фінансах, освіті, автоматизації рутинних задач та створенні прототипів.
- Швидкий поріг входження для новачків. Завдяки простому синтаксису Python часто обирається як перша мова для вивчення програмування в академічному середовищі. [15]

**Основні недоліки Python:**

- Відносно низька швидкодія. Як інтерпретована мова, Python повільніший за компільовані мови, такі як C++, Java або Rust. Це особливо помітно в задачах, що потребують великої кількості обчислень.
- Проблеми з багатопоточністю. Через наявність глобального інтерпретаторного замка (GIL) Python має обмеження у реалізації ефективного паралельного виконання потоків у багатоядерних процесорах.
- Менш зручний для створення мобільних застосунків. Порівняно з Java або Kotlin, екосистема Python для мобільної розробки значно слабше розвинена.

- Динамічна типізація. Хоча це є перевагою при швидкому прототипуванні, у великих проєктах відсутність суворої типізації може ускладнювати контроль за якістю коду.

**Приклади відомих програм і сервісів, розроблених з використанням Python:**

- Instagram — серверна частина платформи реалізована на фреймворку Django.
- YouTube — окремі внутрішні інструменти Google, що використовуються для підтримки відеоплатформи, створені на Python.
- Spotify — алгоритми рекомендацій та аналітики слухацької поведінки реалізовані з використанням Python.
- Dropbox — клієнтські та серверні компоненти побудовані на Python, що дозволило досягти кросплатформеності.
- Reddit — спочатку повністю написаний на Python, підтримує високонавантажену соціальну платформу.
- Pinterest — активно використовує Python у системах обробки зображень та обслуговування користувацьких запитів.
- NASA — застосовує Python для аналізу супутникових даних, моделювання та досліджень у сфері астрономії.

Python залишається однією з найпопулярніших мов програмування у світі. Висока продуктивність розробки, гнучкість і велика кількість готових рішень роблять її доцільним вибором для широкого спектра завдань у галузі інформаційних технологій

### 2.2 JavaScript

JavaScript — це мова програмування, яка є стандартом для реалізації клієнтської логіки у веб-застосунках. Вона була створена у 1995 році компанією Netscape Communications і спочатку призначалася для додавання інтерактивності до вебсторінок. З часом JavaScript перетворився з простої скриптової мови на повноцінну багатопарадигмальну платформу розробки, яка підтримує об'єктно-орієнтоване, функціональне та реактивне програмування.

Мова стала ключовим елементом технологічного стеку HTML + CSS + JavaScript, який є основою будь-якого сучасного веб-інтерфейсу. JavaScript виконується безпосередньо у браузері, що дає змогу створювати динамічні інтерфейси, реалізовувати реакцію на дії користувача, обмінюватися даними з сервером без перезавантаження сторінки (через технології AJAX, Fetch API) тощо.

Сучасний JavaScript активно підтримує стандарти ECMAScript. Останні версії мови включають класи, модулі, проміси, асинхронні функції, стрілкові функції, колекції та інші засоби, які значно підвищують її виразність і гнучкість. [16]

**Переваги JavaScript:**

- **Вбудована підтримка в усіх сучасних браузерах.** Не потребує встановлення додаткових плагінів або середовищ виконання.
- **Виконується на стороні клієнта.** Забезпечує низьку затримку при взаємодії з інтерфейсом і не потребує постійних запитів до сервера для обробки кожної дії користувача.
- **Підтримка асинхронного програмування.** Завдяки функціям async/await, об’єктам Promise, та подієвій моделі, JavaScript дозволяє ефективно працювати з асинхронними запитами до API.
- **Розвинена екосистема**. JavaScript має найбільшу кількість сторонніх

бібліотек і фреймворків серед усіх мов, доступних через менеджер пакетів npm.

- **Широке застосування поза браузером.** За допомогою середовища виконання Node.js JavaScript активно використовується також для розробки серверної логіки, інструментів командного рядка, десктопних і навіть мобільних застосунків.
- **Підтримка сучасних концепцій UI.** JavaScript дозволяє створювати односторінкові застосунки (SPA), реактивні інтерфейси, інтерактивні компоненти та інші складні елементи сучасного вебу.

**Недоліки JavaScript:**

- **Відсутність суворої типізації.** Через динамічну природу мови можуть виникати помилки, які виявляються лише під час виконання програми, що ускладнює її тестування та масштабування.
- **Різна поведінка в різних браузерах.** Незважаючи на стандартизацію, окремі браузери можуть по-різному інтерпретувати певні особливості мови або DOM API.
- **Безпекові ризики.** JavaScript виконується на стороні клієнта, що робить його потенційно вразливим до атак типу XSS (cross-site scripting) або CSRF, якщо не застосовуються відповідні заходи безпеки.
- **Складність у підтримці великих проєктів.** У розгалужених клієнтських застосунках без використання фреймворків або систем типізації (на кшталт TypeScript) код JavaScript може стати важким для обслуговування.
- **Високе навантаження на браузер**. Розробник повинен враховувати продуктивність на стороні користувача, особливо на мобільних пристроях або при великій кількості елементів інтерфейсу.

JavaScript залишається ключовою мовою для клієнтської веб-розробки та є обов'язковим компонентом технологічного стеку для створення повноцінних інтерактивних веб-застосунків.

### 2.3 FastAPI

FastAPI — це сучасний веб-фреймворк для розробки RESTful API, створений Себастьяном Раміресом у 2018 році. Він базується на стандарті ASGI (Asynchronous Server Gateway Interface) і побудований поверх бібліотек Starlette (для обробки запитів) та Pydantic (для валідації даних). FastAPI є асинхронним фреймворком і підтримує одночасно синхронне й асинхронне виконання функцій, що робить його особливо ефективним у застосунках із високою частотою запитів, паралельною обробкою та взаємодією з зовнішніми API.

Фреймворк швидко набув популярності серед розробників завдяки своїй продуктивності, простоті використання та активному розвитку. Він офіційно рекомендується компаніями Netflix, Microsoft, Uber, Amazon, Zulip та іншими для створення API-сервісів. [17]

**Переваги FastAPI:**

- **Висока продуктивність.** FastAPI побудований з урахуванням асинхронного виконання та використовує неблокуючу архітектуру. Завдяки цьому продуктивність фреймворку наближається до таких мов як Go і Node.js. У тестах він демонструє одну з найвищих швидкостей обробки HTTP-запитів серед фреймворків на Python.
- **Автоматична генерація документації.** Фреймворк автоматично створює документацію API у форматі OpenAPI (Swagger UI та ReDoc) на основі анотацій типів у функціях. Це суттєво спрощує розробку, тестування та інтеграцію з іншими системами.
- **Типізованість і інтеграція з Pydantic.** FastAPI підтримує сувору

типізацію аргументів і відповідей за допомогою анотацій типів Python та моделей pydantic. Це забезпечує автоматичну перевірку вхідних даних, їх серіалізацію/десеріалізацію та формування помилок без додаткового коду.

- **Асинхронність.** Фреймворк повноцінно підтримує асинхронні функції (async def), що дає змогу обробляти тисячі запитів одночасно без блокування потоку. Це особливо важливо при роботі з мережевими запитами або базами даних.
- **Зручність структурування коду.** FastAPI дозволяє організовувати код у вигляді модулів з розділенням на маршрути, моделі, сервіси, що відповідає принципам чистої архітектури та полегшує підтримку проєкту.
- **Інтуїтивно зрозумілий синтаксис.** Фреймворк має чіткий, лаконічний синтаксис, що спрощує навчання та скорочує кількість шаблонного коду.
- **Активна спільнота та підтримка.** FastAPI має розгалужену документацію, постійно оновлюється та має широке коло користувачів. Його часто застосовують у навчальних курсах, стартапах і промислових проєктах.

**Недоліки FastAPI:**

- **Висока залежність від типів.** FastAPI активно використовує типізацію Python, що, з одного боку, підвищує надійність, а з іншого — потребує суворого дотримання правил типізації, що може ускладнювати розробку для початківців або при роботі зі слабо структурованими даними.
- **Обмежена кількість готових рішень.** Порівняно з Django, у FastAPI немає вбудованої ORM, панелі адміністратора, шаблонізатора або

автентифікації «з коробки». Це змушує розробника самостійно обирати відповідні бібліотеки для реалізації базових функцій.

- **Вища складність для розгортання на деяких хостингах.** Оскільки FastAPI побудований на ASGI, його потрібно запускати через Uvicorn або Gunicorn з ASGI-воркером, що іноді вимагає додаткової конфігурації на хостингу.
- **Недостатня сумісність зі старими версіями Python.** FastAPI вимагає Python 3.7 або вище, тому не може бути використаний у застарілих середовищах.
- **Молодість фреймворку.** Порівняно з Flask чи Django, FastAPI є значно молодшим, а отже має менше прикладів використання в складних комерційних системах та менше готових розширень.

FastAPI є сучасним інструментом, який відповідає актуальним вимогам до створення високопродуктивних API-сервісів. Завдяки підтримці асинхронності, типізації, інтеграції з Pydantic та генерації документації, він є доцільним вибором для створення швидких і масштабованих веб-застосунків на Python.

### 2.4 IDE PyCharm

Для розробки дипломного проекту було використано програмне середовище PyCharm, яке є одним із найпопулярніших для розробки на Python.

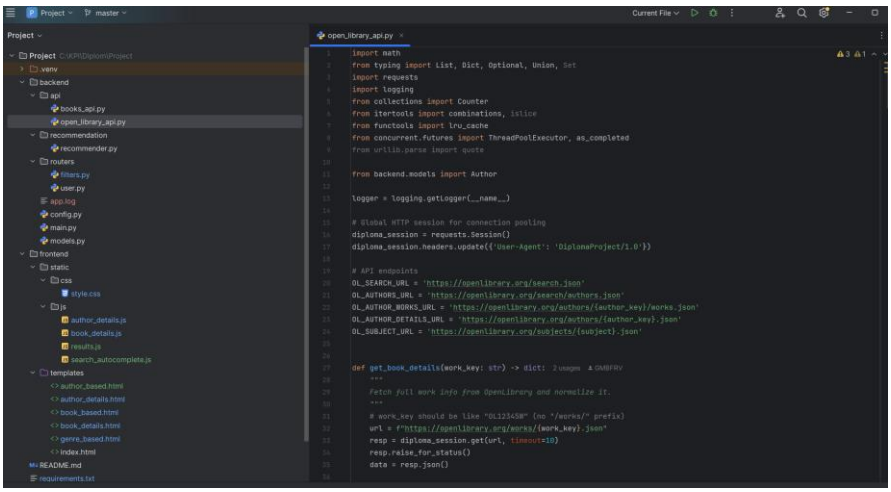



Рисунок 2.1 – IDE PyCharm

IDE PyCharm забезпечує комфортну розробку та цілком підходить для реалізації веб-застосунку за темою дипломного проекту, до основних переваг цього середовища можна віднести наступне:

- Автоматичне виділення синтаксичних помилок
- Зручна інтеграція із системами контролю версій, в тому числі Git
- Можливість швидкого створення і простого використання віртуального середовища

# ВИСНОВОК ДО РОЗДІЛУ 2

У даному розділі було розглянуто основні інструменти та технології, використані при реалізації дипломного проєкту, розглянуто їх переваги та недіоліки, обгрунтовано вибір конкретних технологій для проведення розробки.

Для розробки клієнтської частини вирішено використати HTML для створення шаблонів сторінок, JavaScript для створення скриптів для них та забезпечення взаємодії із користувачем і CSS для створення стилів і оформлення інтерфейсу веб-застосунку.

Для розробки серверної частини використано мову програмування Python і фреймворк FastAPI який дозволяє зручно налаштовувати маршрутизацію та розробляти ефективні застосунки із високою частотою запитів.

У якості джерела даних використано Open Library API для основної інформації, вибір обгрунтовано можливістю доступу без реєстрації, відсутністю лімітів на кількість запитів, зручні параметри при побудові запитів які дозволяють гнучко взаємодіяти з API, а також велику кількість доступних книг і авторів. Як додаткове джерело використано Wikipedia API, яке використовується виключно для отримання опису об’єкту, причиною його застосування є низька якість або взагалі відсутність описів у OpenLibrary API.

Розробка буде проводитися у IDE PyCharm, яке надає можливість зручної інтеграції з системами контролю версій, дозволяє швидко створювати віртуальне середовище, та містить автоматичні перевірки для визначення помилок, що дозволяє зручно та швидко розробляти програмні продукти.

# РОЗДІЛ 3 РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

Цей розділ містить детальний огляд реалізації веб-застосунку та його структури, розроблених програмних компонентів та їх призначення, особливостей взаємодії окремих функцій та класів. Розглянуто механізм роботи застосунку, в тому числі особливості та основні принципи взаємодії з користувачем, API, створення рекомендацій та їх вивід.

## 3.1 Архітектура веб-застосунку

Для розробки застосунку використано принципи REST API, а відповідно зв'язок між клієнтською та серверною частинами забезпечується шляхом надсилання HTTP-запитів.

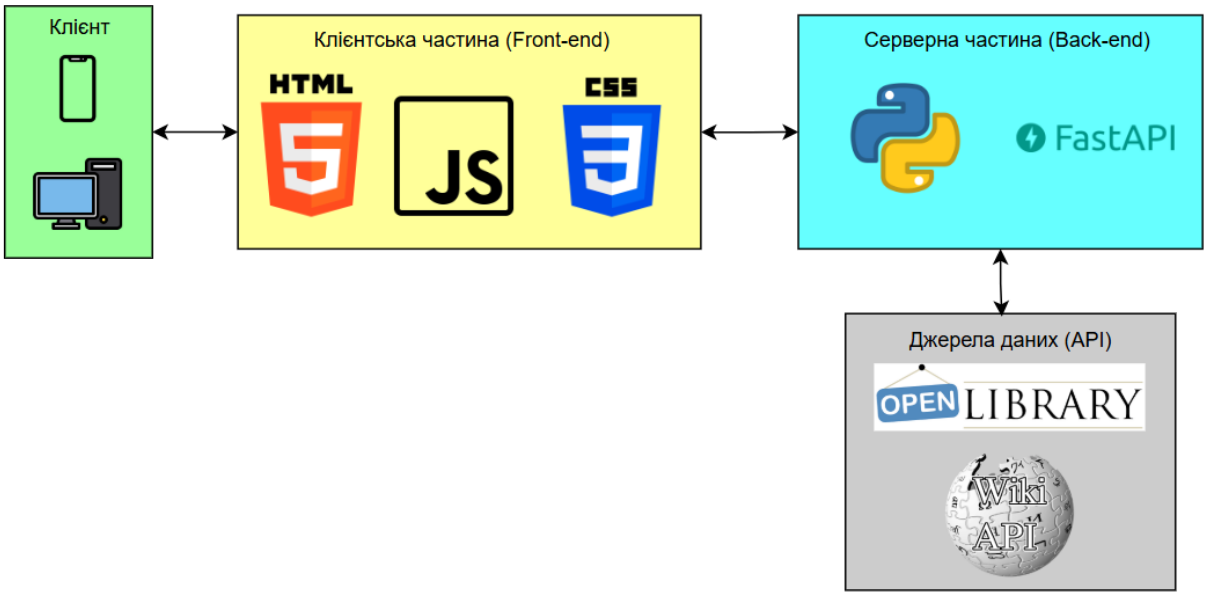



Рисунок 3.1 – Архітектура веб-застосунку

Після отримання запиту від користувача на клієнтській частині веб-застосунку, він надсилає запит до серверної частини для отримання списку рекомендацій. В свою чергу, серверна частина надсилає запит до OpenLibrary API для отримання основної частини інформації і за необхідністю до Wikipedia API для отримання опису книги або біографії автора (через низьку якість цих

даних у Open Library API). Після отримання даних від API спрацьовує рекомендаційна система, яка будує список рекомендацій та повертає його на клієнтську частину веб-застосунку, який в свою чергу виводить його користувачеві.

### 3.2 Клієнтська частина веб-застосунку

Для реалізації клієнтської частини веб-застосунку було вирішено створити окрему директорію у проекті, що буде містити створення HTML-шаблонів, JS скрипти для обробки користувацьких дій на них, а також файлу зі стилями для створення сучасного, інтуїтивно зрозумілого і клієнтоорієнтованого інтерфейсу.

Структура клієнтської частини веб-застосунку наведена на рисунку нижче:

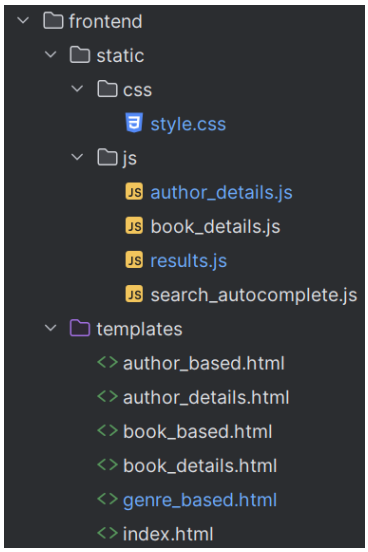



Рис 3.1 – Структура клієнтської частини веб-застосунку

Використання наведеної структури дозволяє легко масштабувати та модифікувати проект, а також полегшує розуміння його структури та призначення окремих компонентів.

Основні функції клієнтської частини проекту:

- **Генерація шаблонів:** генерація HTML-сторінок із використанням та

стилізація за допомогою CSS.

- **Обробка подій:** перехоплення кліків, введення тексту й інших дій користувача за допомогою JavaScript-модулів (наприклад, search_autocomplete.js, results.js).
- **Збір і валідація даних:** отримання значень із форм і полів введення, перевірка коректності перед відправкою на сервер.
- **Асинхронна комунікація:** відправка HTTP-запитів до бекенду через Fetch API, обробка JSON-відповідей.
- **Динамічне оновлення вмісту:** підвантаження результатів без перезавантаження сторінки (автодоповнення запитів, підгрузка списків рекомендацій).
- **Навігація:** перемикання між головною видачею та детальними сторінками книги або автора, підтримка «хлібних крихт» та історії браузера.
- **Обробка станів і помилок:** відображення індикаторів завантаження, повідомлень про помилки мережі чи відсутність результатів.

### 3.1.1 Маршрути

Усі шляхи запиту у застосунку реалізовано відповідно до архітектури REST API для обробки запитів типу GET, тобто запитів для отримання інформації, для передачі інформації використовується формат даних JSON. Також реалізовано автоматичну валідацію параметрів за допомогою бібліотеки Pydantic.

Основні маршрути та їх призначення:

- **GET /api/autocomplete?q={query}**

  Використовує функцію search_autocomplete(query: str) – запит на отримання списку підказок за фрагментом тексту, введеним

користувачем.

Реалізовано для автодоповнення в полі пошуку. Повертає масив рядків, що відповідають початку імен книг або авторів. Запит виконується асинхронно під час введення. Валідація: рядок має містити не менше 3 символів.

- **GET /api/search?query={query}**

  Використовує функцію search_books(query: str) або search_authors(query: str) – універсальний запит для пошуку за введеним запитом.

  Залежно від вмісту відповіді, система повертає перелік книг, авторів або обидва типи об’єктів. Результати ранжуються за релевантністю або повнотою відповідності.

- **GET /api/book/{olid}**

  Використовує функцію get_book_details(olid: str) – отримання повної інформації про книгу за її унікальним ідентифікатором OpenLibrary. Відповідь містить назву книги, список авторів, опис, рік публікації, жанрову приналежність, рейтинг, ідентифікатор обкладинки тощо. Отримані дані використовуються для формування детальної сторінки книги.

- **GET /api/author/{author_key}**

  Використовує функцію get_author_details(author_key: str) – отримання біографічної та бібліографічної інформації про автора.

  Повертається ім’я автора, коротка біографія, список найвідоміших творів, кількість публікацій, релевантні теми, з якими асоціюється його творчість.

- **GET /api/recommend/genre/{genre}?limit={n}**

  Використовує функцію Recommender.recommend_by_genre(genre: str, limit: int) – формування рекомендацій на основі жанру.

Запит надсилається при виборі жанру. Проводиться фільтрація за предметами (subjects) книги. Повертається перелік книг, що найкраще відповідають вказаній жанровій категорії.

- **GET /api/recommend/similar/{olid}?limit={n}**
  Використовує функцію Recommender.recommend_similar_books(olid: str, limit: int) – пошук книг, схожих на вказану за змістом або тематикою.
  Опис цільової книги обробляється через алгоритм TF-IDF. Далі обчислюється косинусна подібність до інших книг. Повертаються ті твори, чий зміст найбільше співвідноситься з обраним.
- **GET /api/recommend/author/{author_key}?limit={n} (за наявності)**
  Використовує функцію
  Recommender.recommend_similar_authors(author_key: str, limit: int) – пошук авторів, схожих на заданого.
  Профіль автора будується на основі його тематичних категорій.
  Знаходяться інші автори з найбільшим тематичним перетином.
  Повертається список схожих авторів із базовою інформацією.

### 3.1.2 HTML-шаблони

Шаблони HTML у веб-застосунку використовуються для формування структури сторінок, що надсилаються клієнту з серверної частини. Усі шаблони реалізовані за допомогою системи шаблонів Jinja2, яка дозволяє вставляти динамічні дані у структуру HTML-коду. Шаблони рендеряться сервером, і на основі вхідних даних формуються веб-сторінки з відповідним вмістом.

У межах застосунку реалізовано декілька окремих HTML-шаблонів, кожен з яких відповідає за відображення окремої частини функціональності:

- **index.html**

  Головна сторінка сервісу «Book Discovery Service»: містить заголовок, короткий опис переваг (Genre Explorer, Author Match, Book Companion) та секцію «Start Discovering» із трьома кнопками-посиланнями для переходу до функціоналу за жанром, автором або книгою .

- **genre_based.html**

  Сторінка вибору жанрів (шаблон для маршруту /genre_filter): у шапці є кнопка «← Back» і заголовок «Select Your Favorite Genres», далі набір кнопок із жанрами (.genre-btn) та кнопка «Find Books», а під ними — контейнер div#results для відображення результатів. Після кліку «Find Books» блок results заповнюється через results.js .

- **book_based.html**

  Сторінка пошуку схожих книг (шаблон для /book_filter): у шапці кнопка «← Back» і заголовок «Find Similar Books», поле введення назви (input#bookTitleInput) із випадаючим списком підказок (div#bookSuggestions) та кнопка «Search». Після запиту результати відображаються в контейнері div#results через results.js і search_autocomplete.js .

- **author_based.html**

  Сторінка пошуку схожих авторів (шаблон для /author_filter): містить кнопку «← Back», заголовок «Find Similar Authors», поле введення (input#authorInput) з випадаючим меню підказок (div#authorSuggestions) і кнопку «Search», а також блок div#authorResults для результатів. Логіку реалізують search_autocomplete.js і results.js

- **book_details.html**

  Детальна сторінка книги (/book/{olid}): включає кнопку «← Back», контейнер div#book-detail для метаданих (назва, автори, рік, жанри,

опис, обкладинка) та секцію div#recommendations з картками рекомендованих книг. Вміст генерується модулем book_details.js

- **author_details.html**

  Детальна сторінка автора (/author/{author_key}): містить кнопку «← Back», контейнер div#author-detail для біографії, фото та статистики (дата народження, число робіт, рейтинг) і секцію div#author-books з картками творів автора. Заповнення здійснює author_details.js
- Ці шаблони забезпечують чітку роздільну відповідальність: загальний стиль і навігацію задає index.html, сценарії пошуку - окремі сторінки для жанрів, книг і авторів, а сторінки деталей відображають розширену інформацію з динамічними рекомендаціями.

### 3.1.3 JavaScript-модулі та їх логіка

Для обробки подій, забезпечення динамічної зміни наповнення сторінки, реалізації системи автодоповнення розроблено скрипти мовою програмування JavaScript. Усього проект містить 4 модулі із такими скриптами, кожен з яких забезпечує окрему частину функціоналу.

#### 3.1.3.1 search_autocomplete.js

**Загальне призначення :**

Модуль забезпечує функціонал автодоповнення для полів пошуку книг і авторів. Він прослуховує введення користувача, застосовує затримку для зменшення кількості запитів, звертається до відповідних ендпоїнтів і відображає список підказок у випадаючому списку.

### 3.1.3.2 results.js

**Загальне призначення:**

Даний модуль реалізує базову обробку подій для кожної з базових сторінок, окрім сторінок із детальною інформацією про об’єкт. Містить в собі обробку натискань кнопок навігації та пошуку та вивід рекомендацій після запиту для відповідних сторінок.

**Ключові функції модуля:**

- **toggleGenreSelection(event)** – відмічає або знімає жанр із обраних, оновлює вигляд кнопки.
- **performGenreSearch()** – формує та відправляє запит до /api/genre_filter, обробляє відповідь і викликає рендеринг карток книг.
- **performAuthorSearch()** – формує та відправляє запит до /api/author?author=…, обробляє відповідь і викликає рендеринг карток авторів.
- **performSearch()** - формує та відправляє запит до /api/book?book=…, обробляє відповідь і викликає рендеринг карток книг.

### 3.1.3.3 book_details.js

**Загальне призначення:**

Модуль відповідає за побудову детальної сторінки книги. Він отримує ідентифікатор виду OLID з URL, завантажує дані про книгу та список схожих творів, а потім відображає цю інформацію у відповідних секціях сторінки.

**Ключові функції модуля:**

- **initBookDetails()** – ініціалізує процес завантаження: дістає OLID із шляху сторінки та викликає fetch-функції.
- **fetchBookData(workKey)** – надсилає запит до /api/book/{olid}, отримує метадані й рекомендації.

- **renderBookDetail(detail)** – виводить назву, авторів, опис, жанри та обкладинку у блок #book-detail.
- **renderBookRecommendations(recommendations)** – створює картки рекомендованих книг у секції #recommendations.

#### 3.1.3.4 author_details.js

**Загальне призначення:**

Модуль формує детальну сторінку автора. Він аналізує URL, завантажує біографію, статистику й опис автора, а також, за наявності, список його творів чи рекомендацій, та відображає їх у відповідних секціях.

**Ключові функції модуля:**

- **initAuthorDetails()** – отримує ключ автора з URL та запускає послідовність завантаження даних
- **fetchAuthorData(authorKey)** – надсилає запит до /api/author/{author_key}, отримує біографію та метадані
- **renderAuthorDetail(author)** – відображає ім’я, фото, дати життя та біографічний опис у блоці #author-detail
- **renderAuthorBooks(books)** – створює й відображає картки творів автора у секції #authorBooks.

### 3.1.4 Динамічне оновлення інтерфейсу

Механізми динамічного оновлення інтерфейсу покликані забезпечити користувачеві плавний досвід без повного перезавантаження сторінки. Уся логіка реалізована у JavaScript-модулях із використанням маніпуляцій DOM та зміни CSS-класів.

**Призначення:**

Забезпечити відображення станів «завантаження», «порожньо» та «помилка», а також плавне підвантаження й оновлення результатів на сторінках пошуку та деталей.

**Ключові дії та відповідні функції:**

- **Показ спінера / індикатора завантаження**
  Викликається до початку асинхронного запиту; відображає елемент зі CSS-класом .loading.
  Функція: showLoadingIndicator()
- **Приховання спінера**
  Викликається після отримання відповіді або помилки; ховає індикатор.
  Функція: hideLoadingIndicator()
- **Очищення контейнера результатів**
  Перед новим рендером обирає контейнер (наприклад, #results, #recommendations) і видаляє його вміст.
  Функція: clearResultsContainer(containerId)
- **Відображення повідомлення «Нічого не знайдено»**
  Якщо масив результатів порожній, вставляє елемент із текстом у контейнер та CSS-класом .empty.
  Функція: renderEmptyState(containerId)
- **Відображення повідомлення про помилку**
  У разі збою мережі або некоректної відповіді серверу показує у контейнері елемент з класом .error і поясненням.
  Функція: renderErrorState(containerId, message)

- **Рендеринг карток результатів**

  Після успішного запиту викликає універсальний рендерер із модуля results.js, який створює картки та додає їх до контейнера.

  Функція: appendResultCards(containerId, items)
- **Плавний скрол або фокус**

  Після оновлення результатів здійснює прокрутку до початку списку або встановлює фокус на першу картку.

  Функція: scrollToTop(containerId)

**Сценарії оновлення:**

1. Користувач ініціює пошук або обирає жанр → викликається showLoadingIndicator() → асинхронний запит → при успіху hideLoadingIndicator(), clearResultsContainer(), appendResultCards() (або renderEmptyState()) → scrollToTop().
2. На деталях книги/автора при завантаженні метаданих і рекомендацій спочергово викликаються showLoadingIndicator() та hideLoadingIndicator(), а вміст контейнерів оновлюється без перезавантаження сторінки.
3. У разі помилки мережі або некоректної відповіді — hideLoadingIndicator() → renderErrorState().

Цей набір функцій і сценаріїв гарантує, що користувач завжди бачить актуальний стан запитів, швидко отримує візуальний фідбек і може продовжувати взаємодію з інтерфейсом без зайвої паузи або перезавантаження сторінки.

### 3.2 Серверна частина веб-застосунку

Серверна частина веб-застосунку реалізує повний цикл взаємодії з API, валідацію та перевірку отриманих даних, їх подальшу обробку, фільтрацію та сортування, містить рекомендаційну систему що на основі отриманих даних створює рекомендації відповідно до запиту користувача.

Детальні відомості щодо основних модулів серверної частини застосунку наведені у відповідних підпунктах.

#### 3.2.1 Архітектура та модульна організація

Сервісна частина веб-застосунку має наступний вигляд:

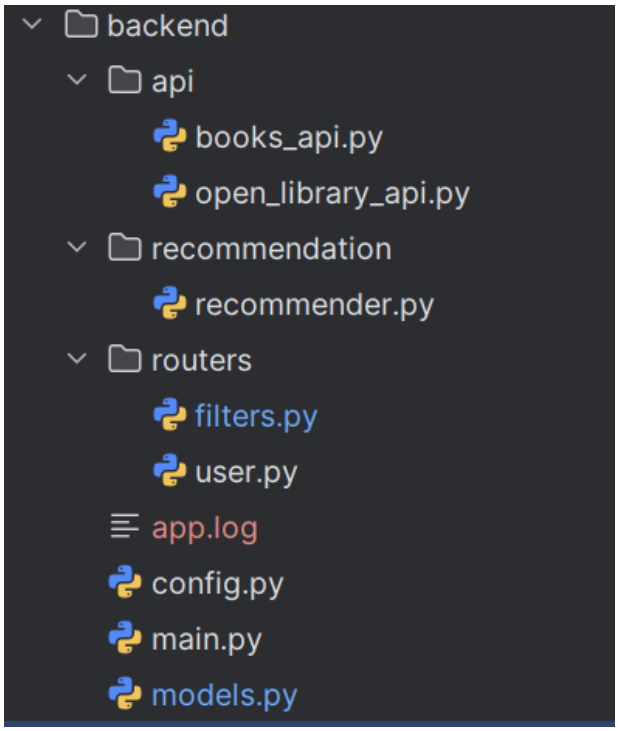



Рисунок 3.2 – Структура сервісної частини веб-застосунку

Для забезпечення логічного поділу відповідальностей проєкт структуровано на модулі за функціональним призначенням.

Основні компоненти організовано наступним чином:

- **main.py** – головний файл застосунку. Ініціалізує FastAPI-об'єкт, підключає роутери, налаштовує CORS та конфігурацію запуску.

- **models.py** – містить Pydantic-моделі для опису структури відповідей API (Book, Author, BookDetailSchema).
- **filters.py** – реалізує пошук книг, авторів, автодоповнення, обробку запитів за жанрами.
- **open_library_api.py** – реалізує функції доступу до OpenLibrary: пошук книг, отримання деталей за ID, запити по авторам.
- **recommender.py** – містить клас Recommender з методами формування рекомендацій на основі жанру, змісту книги або автора. Тут реалізовано використання TF-IDF і cosine similarity для аналізу схожості.

Використання такої структури забезпечує легкість та зручність при внесенні змін і полегшує розширення функціоналу застосунку, а також забезпечує легкість для сприйняття.

### 3.2.2 Маршрутизація HTTP-запитів

Модуль filters.py відповідає за обробку основних користувацьких запитів, пов’язаних з пошуком книг, авторів, фільтрацією за жанрами та автодоповненням. Саме цей файл містить найважливіші REST-ендпоїнти застосунку.

Ключовий функціонал:

- **Пошук книг і авторів за ключовим запитом**
  Реалізовано два окремі маршрути для пошуку книг (/api/search/books) і авторів (/api/search/authors). Залежно від запиту користувача, система виконує відповідний пошук у зовнішньому API OpenLibrary, структурує результати та повертає у форматі JSON.
- **Автодоповнення при введенні тексту**
  Ендпоїнт /api/autocomplete приймає часткове введення користувача та

повертає список найбільш релевантних відповідей (імен авторів або назв книг). Працює на основі результатів OpenLibrary і використовується для формування випадаючих підказок у полі пошуку.

- **Фільтрація за жанрами**
  Маршрут /api/recommend/genre/{genre} виконує фільтрацію на основі переданого жанру. Використовується у сценарії рекомендації книг за тематикою.
- **Пошук схожих книг та авторів**
  Обробляються запити на /api/recommend/similar/{olid} та /api/recommend/author/{author_key}. У першому випадку формується список схожих книг за змістом (на основі TF-IDF), у другому — підбираються схожі автори за тематичним профілем.

### 3.2.3 Взаємодія з API

Файл open_library_api.py відповідає за взаємодію веб-застосунку з зовнішнім сервісом OpenLibrary API, що є основним джерелом інформації про книги, авторів, жанри, обкладинки та описові метадані, а також із Wikipedia API для отримання якісного опису об’єкту. Саме через цей модуль відбувається отримання зовнішніх даних, які надалі використовуються у логіці пошуку, автодоповнення та формування рекомендацій.

Файл реалізує ізольований шар взаємодії із зовнішньою мережею, інкапсулюючи в собі логіку відправлення HTTP-запитів, перевірки статусу відповіді, обробки JSON і відбору лише релевантних полів для подальшої обробки або відображення.

**Основні функції модуля open_library_api.py:**

- **get_book_details(olid: str)**

  Здійснює запит до /books/{olid}.json на OpenLibrary для отримання повного опису книги. Витягує такі дані, як: назва, автор(и), рік публікації, жанри, опис, ідентифікатор обкладинки.

  *Призначення:* використовується для побудови детальної сторінки книги та для аналізу в алгоритмах рекомендації.
- **get_author_details(author_key: str)**

  Надсилає запит до /authors/{author_key}.json і /authors/{author_key}/works.json. Отримує біографію, популярні твори, кількість опублікованих робіт, тематику творчості.

  *Призначення:* забезпечує дані для детальної сторінки автора та формування профілю для рекомендацій.
- **search_books(query: str)**

  Виконує пошук книг за запитом користувача, звертаючись до /search.json?q=…. Повертає список книг, які відповідають критеріям (з назвою, обкладинкою, принаймні одним автором).

  *Призначення:* використовується у загальному пошуку та автодоповненні.
- **search_authors(query: str)**

  Надсилає запит до /search/authors.json?q=…. Повертає авторів, відсортованих за релевантністю та кількістю творів.

  *Призначення:* використовується у пошуку авторів за фрагментами імені або прізвища.
- **get_books_by_subject(subject: str)**

  Здійснює тематичний пошук книг за жанром (subject). Дані витягуються через параметр subject у запиті

  *Призначення:* використовується у сценарії рекомендації за жанром.
- **extract_valid_books(data: list)**

Фільтрує список книг, вилучаючи ті, які не мають необхідної інформації (опису, обкладинки або авторів).

*Призначення:* покращення якості видачі та виключення неповних або технічно порожніх результатів.

### 3.2.4. Рекомендаційна система

Файл recommender.py, розміщений у папці recommendation, містить реалізацію логіки побудови рекомендацій у веб-застосунку. Основна частина функціоналу зосереджена в одному класі — Recommender, який відповідає за формування списків рекомендованих книг або авторів на основі трьох підходів: за жанром, за схожістю змісту книги та за профілем автора.

**Призначення модуля**

Рекомендаційна система використовує як структуровані дані, отримані з OpenLibrary, так і векторні представлення текстів для обчислення схожості між книгами. Основними джерелами для побудови рекомендацій виступають:

- жанри (subjects),
- описи книг (description),
- тематика творчості автора.

Обчислення подібності здійснюється за допомогою алгоритму TF-IDF у поєднанні з косинусною мірою схожості (cosine similarity), що дозволяє оцінювати контекстуальну близькість описів книг.

**Основні функції класу Recommender**

- **recommend_by_genre(genre: str, limit: int = 10)**

  Повертає список книг, що належать до зазначеного жанру (subject).

  *Принцип роботи:* виконується тематичний пошук через OpenLibrary; результати фільтруються, структуруються та сортуються за кількістю

оцінок або популярністю.

*Використовується у сценарії рекомендації за жанром.*

- **recommend_similar_books(olid: str, limit: int = 10)**

  Повертає список книг, зміст яких є подібним до вибраної книги з переданим OLID.

  *Принцип роботи:*

  1. Отримується опис базової книги за її OLID;
  2. Формується корпус з описів інших книг;
  3. Обчислюється матриця TF-IDF;
  4. Вираховується косинусна відстань між базовою книгою та рештою;
  5. Повертається limit найбільш схожих книг.

     *Це основний механізм побудови семантичних рекомендацій.*

- **recommend_similar_authors(author_key: str, limit: int = 10)** *(за наявності реалізації)*

  Повертає список авторів, чия творчість є тематично близькою до зазначеного автора.

  *Принцип роботи:*

  1. Отримується список тем (subjects) творів автора;
  2. Для кожного іншого автора формується тематичний профіль;
  3. Вираховується ступінь тематичного перетину;
  4. Повертаються автори з найвищим рівнем схожості.

**Особливості реалізації**

- Модуль використовує **бібліотеки scikit-learn** (TfidfVectorizer, cosine_similarity) для математичної обробки текстів.
- Список описів книг динамічно формується в процесі обчислень — попереднього збереження корпусу немає, що дозволяє працювати зі свіжими даними в режимі реального часу.

- Кожен результат формується у вигляді словника зі структурованими полями: title, authors, cover_id, rating, description тощо — готовими до передачі фронтенду.

Таким чином, файл recommender.py є ключовим компонентом інтелектуальної частини застосунку. Він забезпечує персоналізацію результатів, підвищує релевантність пошуку та розширює функціональні можливості інтерфейсу користувача. Його реалізація є модульною, гнучкою та придатною для подальшого розширення — наприклад, для впровадження більш складних моделей машинного навчання.

### 3.2.6. Pydantic-моделі (models.py)

Файл models.py містить опис Pydantic-моделей, що використовуються для формалізації структури даних, які обробляються та передаються між бекендом і фронтендом. Pydantic є стандартним інструментом у FastAPI, що забезпечує автоматичну перевірку типів, серіалізацію та десеріалізацію об’єктів Python у формат JSON.

Використання моделей дозволяє централізовано контролювати формат і зміст об’єктів, які відправляються у відповідь на клієнтські запити, а також запобігти помилкам, пов’язаним із неповними або некоректними даними.

**Основні моделі, реалізовані у models.py:**

- **Book**

  Описує базову модель книги у короткому форматі для списку результатів.

  *Основні поля:*

  - key: str – унікальний ідентифікатор книги (OLID);
  - title: str – назва книги;
  - authors: list[str] – список авторів;

- genres: list[str] – список жанрів або тематик;
- rating: float | None – середній рейтинг (опційно);
- rating_count: int | None – кількість оцінок (опційно);
- cover_id: int | None – ідентифікатор обкладинки;
- publish_year: int | None – рік першої публікації.

- **Author**

  Модель для опису автора, що використовується у видачі або в рекомендаціях.

  *Основні поля:*

  - key: str – унікальний ключ автора (OpenLibrary ID);
  - name: str – ім’я автора;
  - top_subjects: list[str] – перелік основних тем у творчості;
  - work_count: int | None – кількість творів;
  - birth_date: str | None – дата народження (опційно);
  - bio: str | None – коротка біографія (опційно);
  - photo_id: int | None – ідентифікатор фото (опціонально для відображення портрету).

- **BookDetailSchema**

  Розширена модель для представлення детальної інформації про книгу. Поля подібні до моделі Book, але включають повний опис (description) і додаткові метадані.

  Використовується для сторінки /book/{olid}.

**Призначення моделей:**

1. **Формалізація структури відповіді API**

   Завдяки Pydantic FastAPI автоматично формує схеми JSON для документації (OpenAPI) і забезпечує відповідність типам.

2. **Валідація вхідних та вихідних даних**

   Перед відправкою відповіді об’єкти приводяться до вказаної структури, а у випадку невідповідності — викидається виняток.

3. **Зручність роботи з даними**

   Моделі надають зрозумілу типізовану структуру, з якою легко працювати при побудові логіки запитів і формуванні відповіді

# ВИСНОВОК ДО РОЗДІЛУ 3

У даному розділі було розглянуто реалізацію клієнтської та серверної частини веб-застосунку, а також структуру проекту.

Використання чіткої структури проекту значно полегшує розуміння принципів його функціонування, а також дозволяє легко масштабувати проект, вносити зміни у конкретні функції або додавати нові, використовувати методи між різними частинами застосунку і виправляти помилки.

Використання модулів написаних мовою програмування JavaScript, HTML-шаблонів сторінок та CSS стилів дозволяє створити сучасний, функціональний та інтуїтивно зрозумілий інтерфейс. В тому числі реалізовано обробку помилок та валідацію введених даних.

Використання FastAPI дозволяє зручно і швидко реалізувати маршрутизацію запитів та їх подальшу обробку.

Реалізація рекомендаційної системи дозволяє їй гнучко підлаштовуватися для створення коректних та актуальних рекомендацій за конкретним критерієм, отриманим від користувача. Функціонал для створення рекомендацій розподілено на окремі функції, що дозволяє легко змінювати параметри конкретних частин системи рекомендацій або додавати новий функціонал при масштабуванні застосунку.

Отже, розроблена структура та програмні модулі забезпечують швидку та гнучку взаємодію як з користувачем, так і з API, містять обробку помилок, валідацію даних, та забезпечують створення та вивід актуальних та доречних рекомендацій із можливістю отримання детальної інформації про них.

# РОЗДІЛ 4 ОГЛЯД ТА ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКУ

Цей розділ містить огляд та тестування розробленого веб-застосунку, в тому числі огляд інтерфейсу, замір часу роботи для основних функцій, аналіз отриманих рекомендацій.

## 4.1 Головна сторінка веб-застосунку

Головна сторінка веб-застосунку містить опис його призначення та функціоналу, а також картки із переадресацією на відповідний фільтр для отримання рекомендацій.

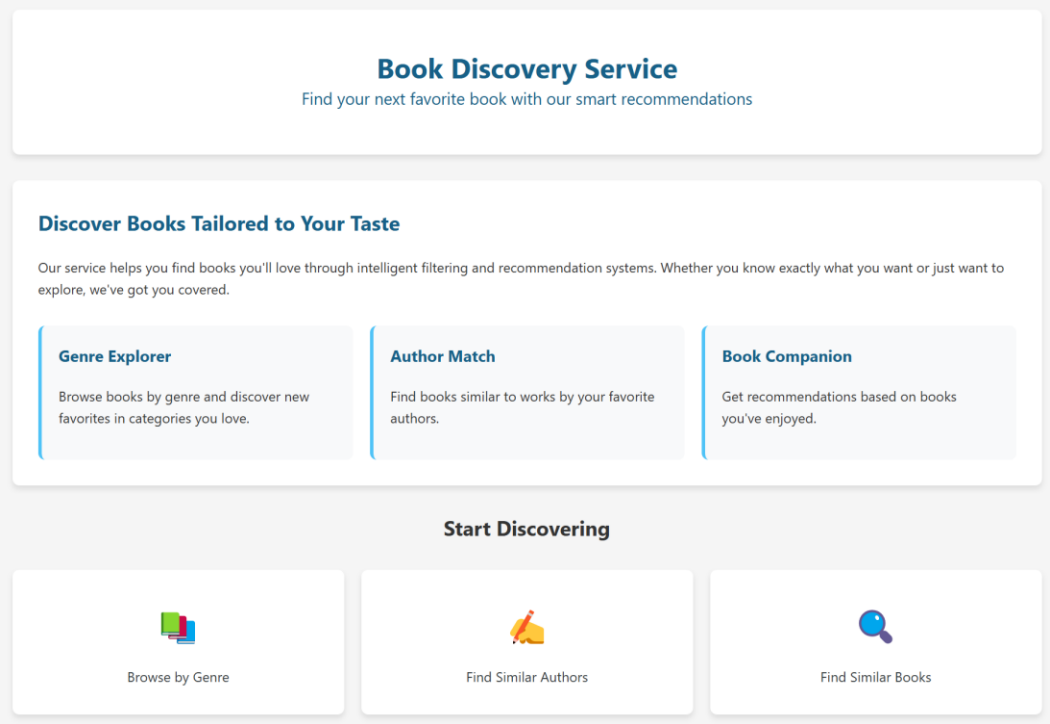



Рисунок 4.1 – Головна сторінка веб-застосунку

## 4.2 Сторінка для пошуку за жанром

Сторінка для пошуку книг за жанром містить кнопки із основними жанрами книг, у користувача є можливість обрати один конкретний жанр або будь-яку їх комбінацію.

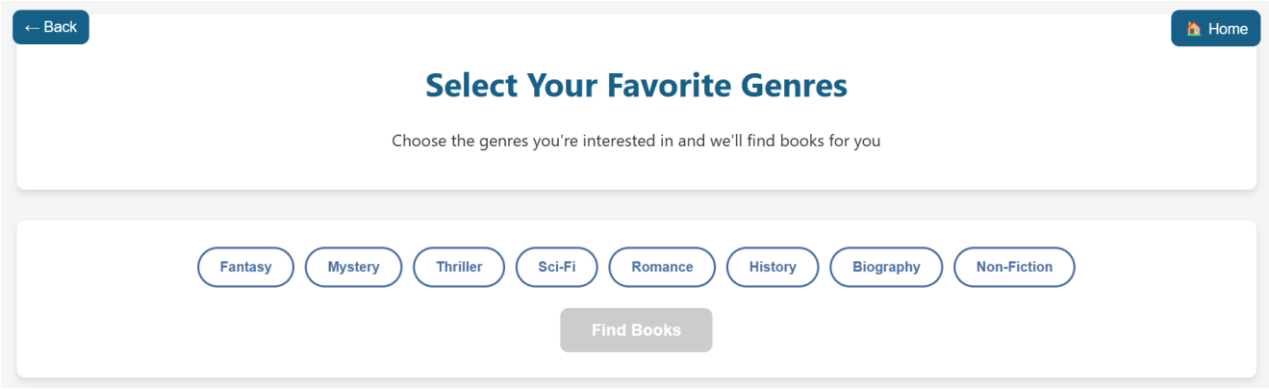



Рисунок 4.2.1 – Сторінка для пошуку книг за жанром

Після вибору жанру(-ів) стає активною кнопка “Find Books”, при натисканні якої користувач отримає добірку книг за відповідним жанром або їх комбінацією.

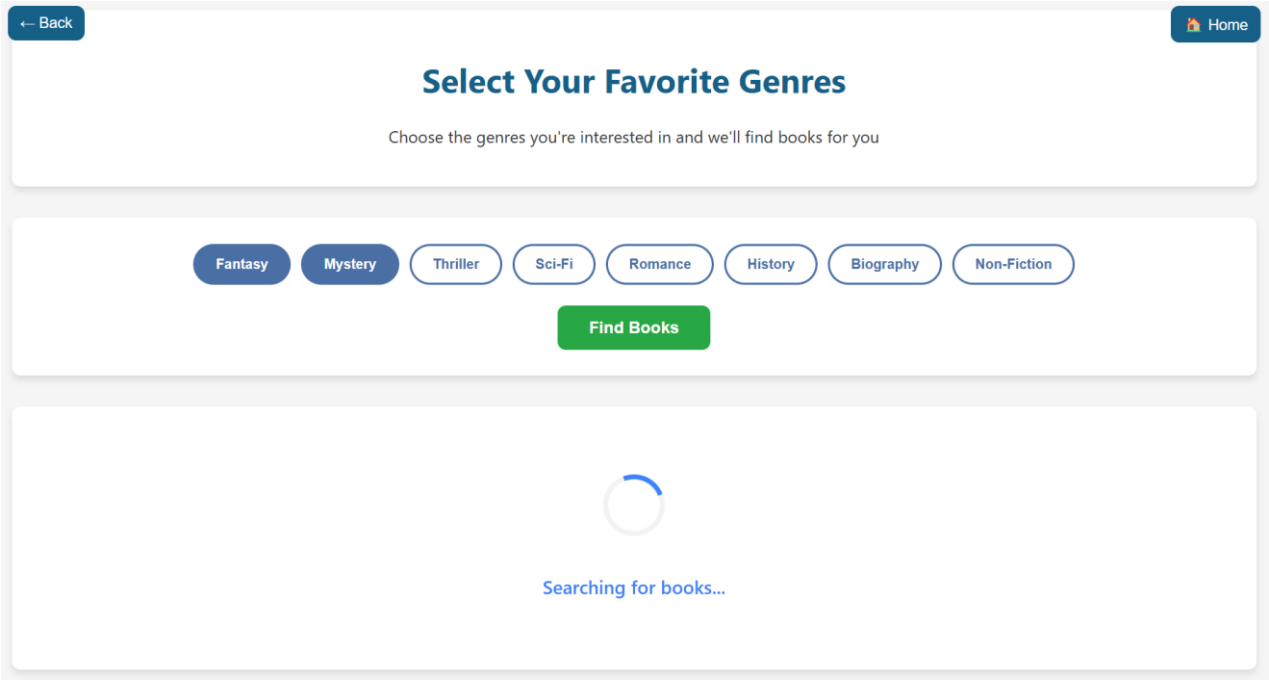



Рисунок 4.2.2 – Сторінка для пошуку книг
за жанром під час завантаження результатів

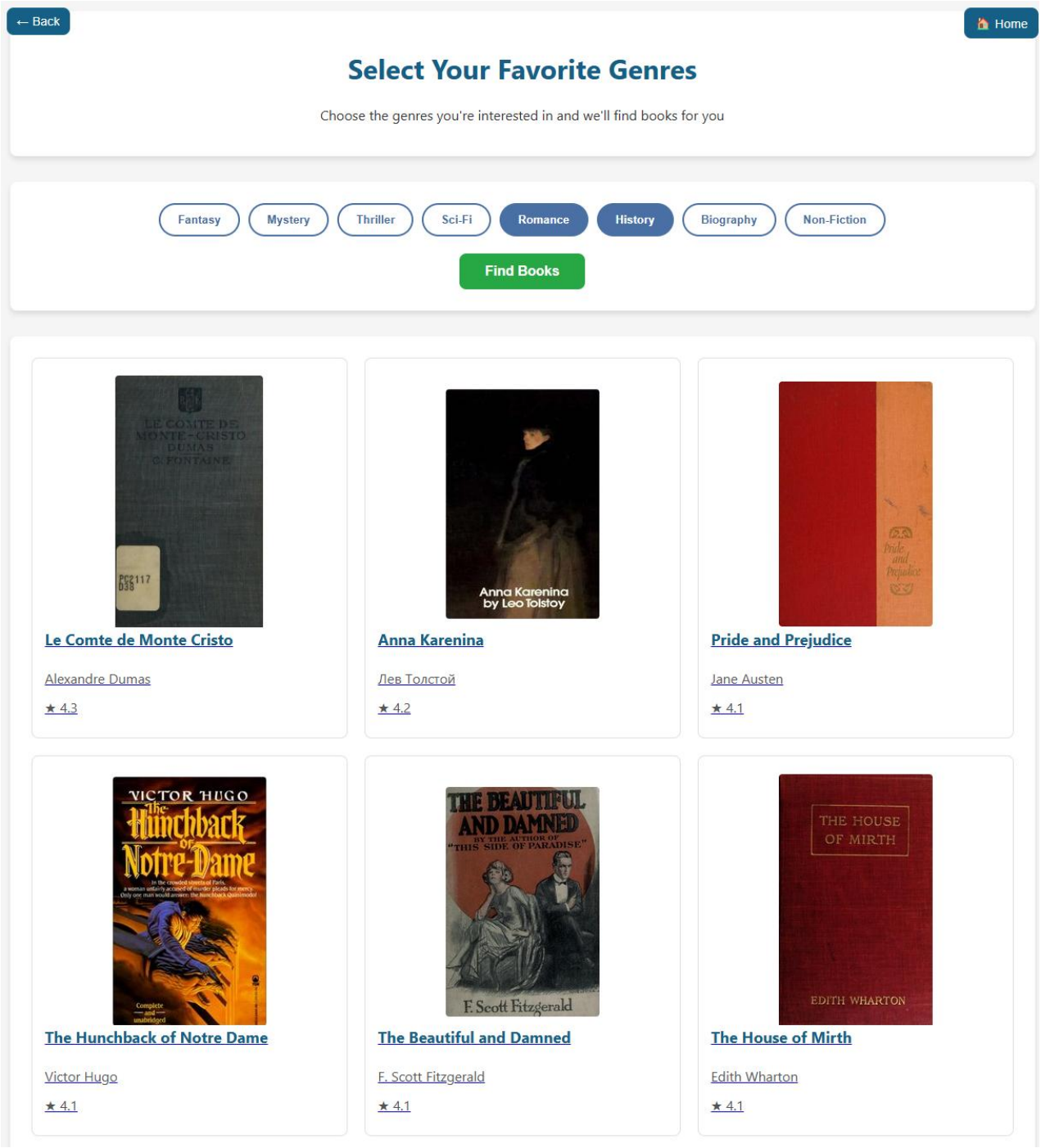



Рисунок 4.2.3 – Рекомендації на основі жанру

Після завершення пошуку та створення рекомендацій користувач має можливість гортати список та завантажувати додаткові результати, а також натиснути на будь-яку з книг для отримання детальної інформації про неї.

### 4.3 Сторінка для пошуку схожих авторів

На початку ця сторінка містить поле для вводу імені автора та кнопку “Search” для здійснення пошуку, причому реалізовано систему підказок.

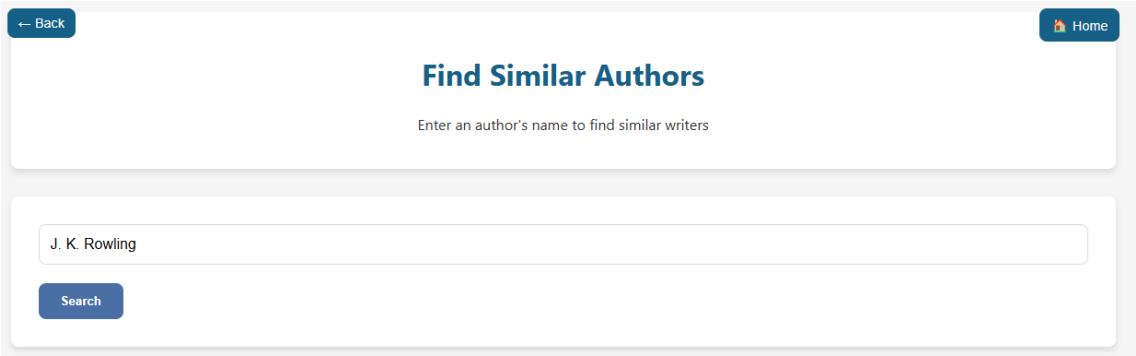



Рисунок 4.3.1 – Сторінка для пошуку схожих авторів


Рисунок 4.3.2 – Система підказок на сторінці для пошуку схожих авторів

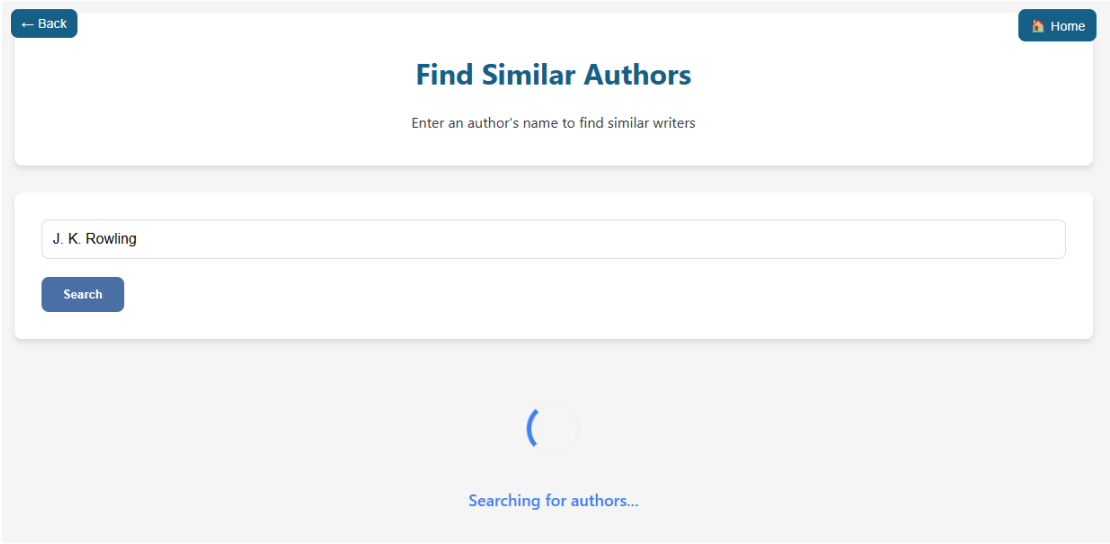



Рисунок 4.3.3 – Сторінка для пошуку схожих авторів під час завантаження рекомендацій

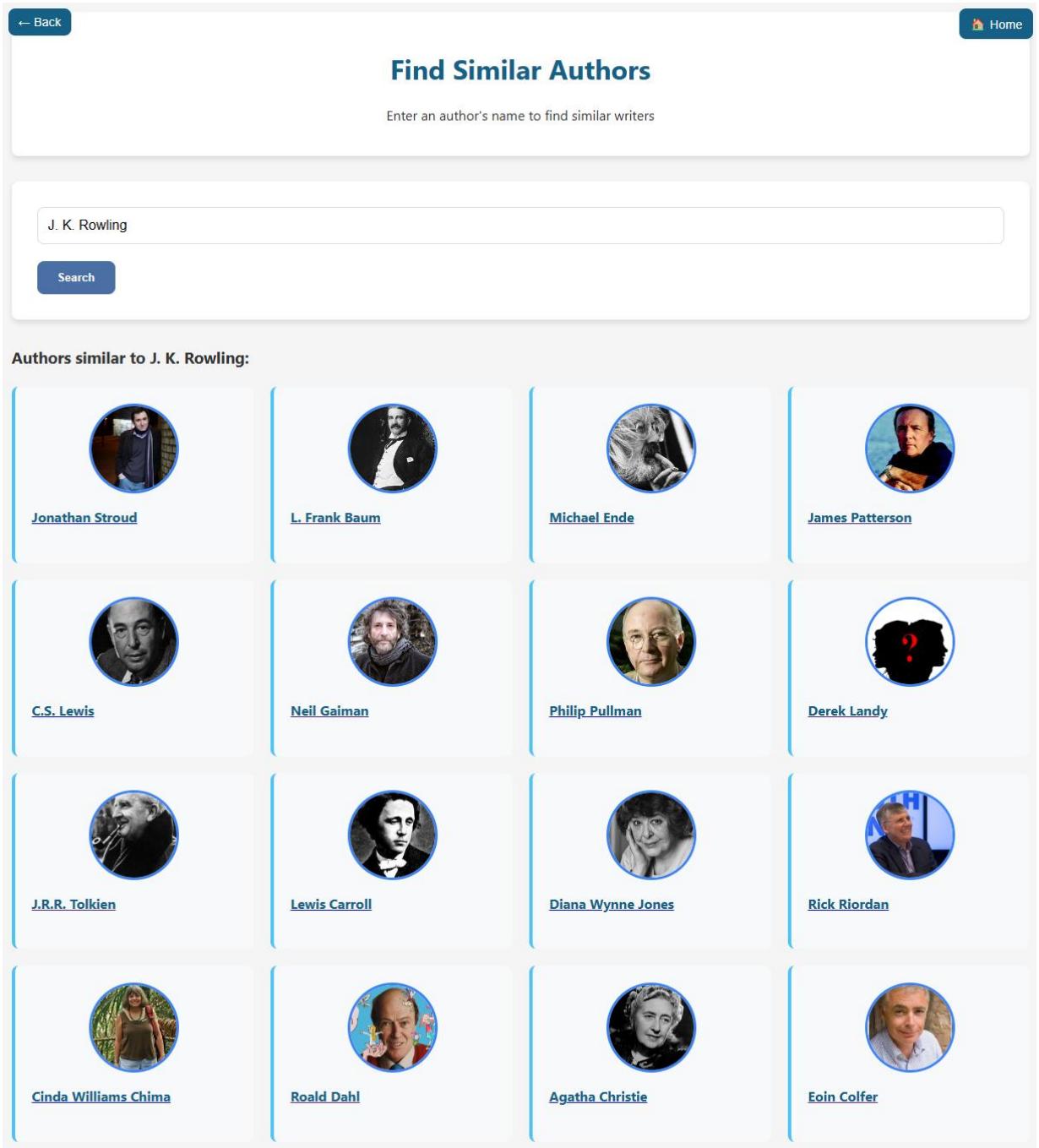



Рисунок 4.3.4 – Підбірка рекомендацій на основі автора

Після завершення пошуку та створення рекомендацій вони виводяться користувачу списком, із можливістю переглянути детальні відомості шляхом натискання на картку будь-якого з авторів.

### 4.4 Сторінка для пошуку схожих книг

Стилістично та функціонально цю сторінку реалізовано аналогічно до сторінки пошуку схожих авторів, адаптовано картки для коректного виводу рекомендованих книг.

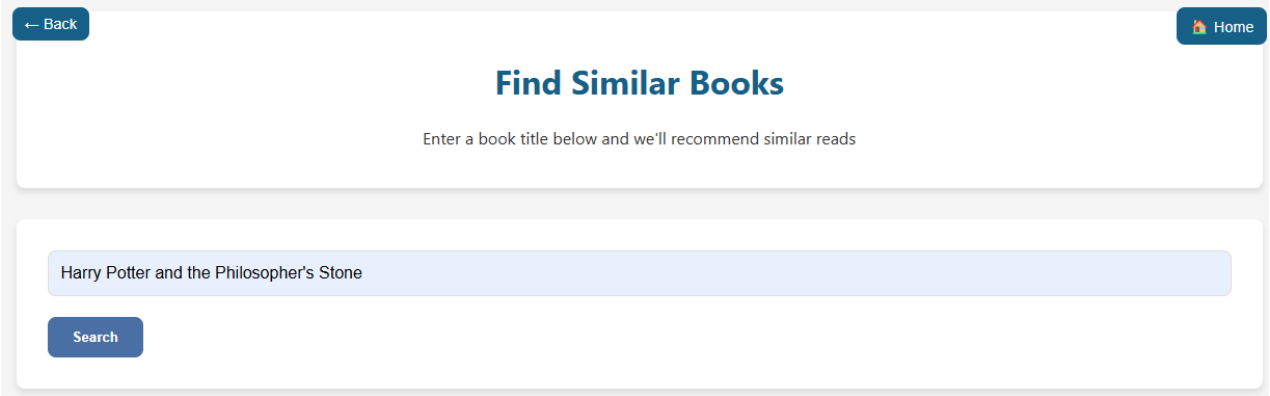



Рисунок 4.4.1 – Сторінка для пошуку схожих книг

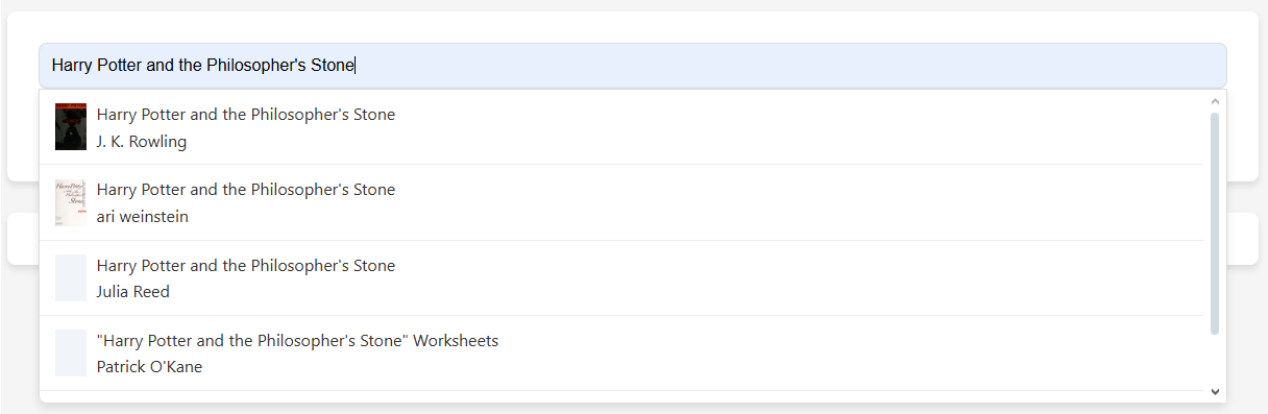



Рисунок 4.4.2 – Система підказок на сторінці для пошуку схожих книг

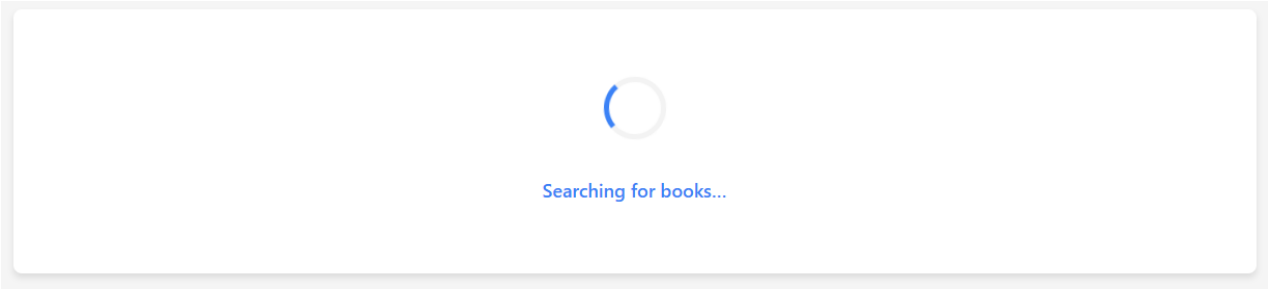



Рисунок 4.4.3 – Графічне відображення процесу завантаження на сторінці для пошуку на основі книги

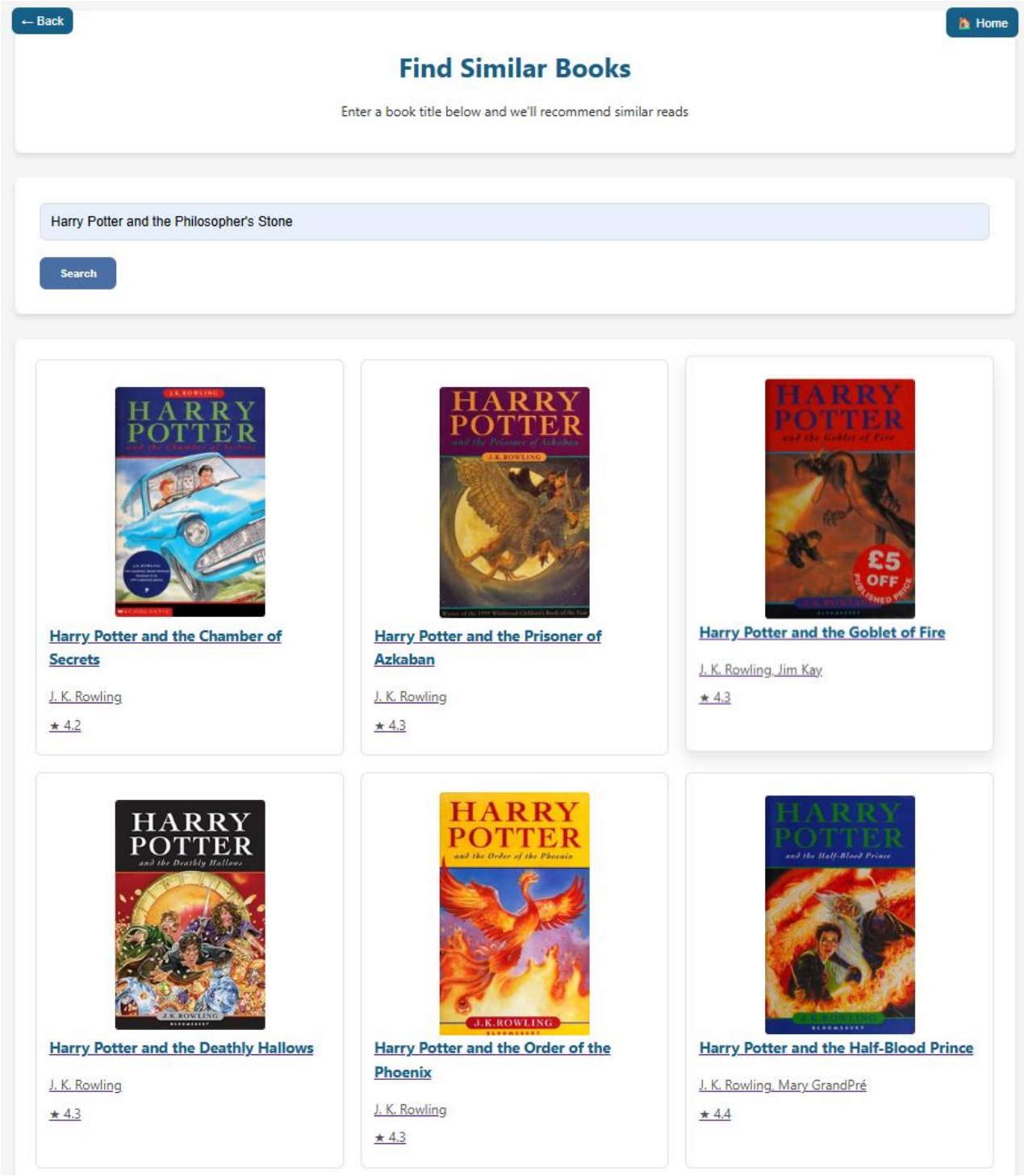



Рисунок 4.4.4 – Вивід рекомендацій на сторінці для пошуку схожих книг

Після виводу рекомендацій, користувач має можливість гортати його та завантажувати додаткові результати, а також переглядати детальну інформацію про книгу при натисканні на її картку.

### 4.5 Перегляд детальної інформації про книгу

Після отримання рекомендацій на основі жанру або книги користувач має можливість натиснути на будь-яку картку книги та отримати детальну інформацію про неї та невелику підбірку рекомендацій схожих книг.

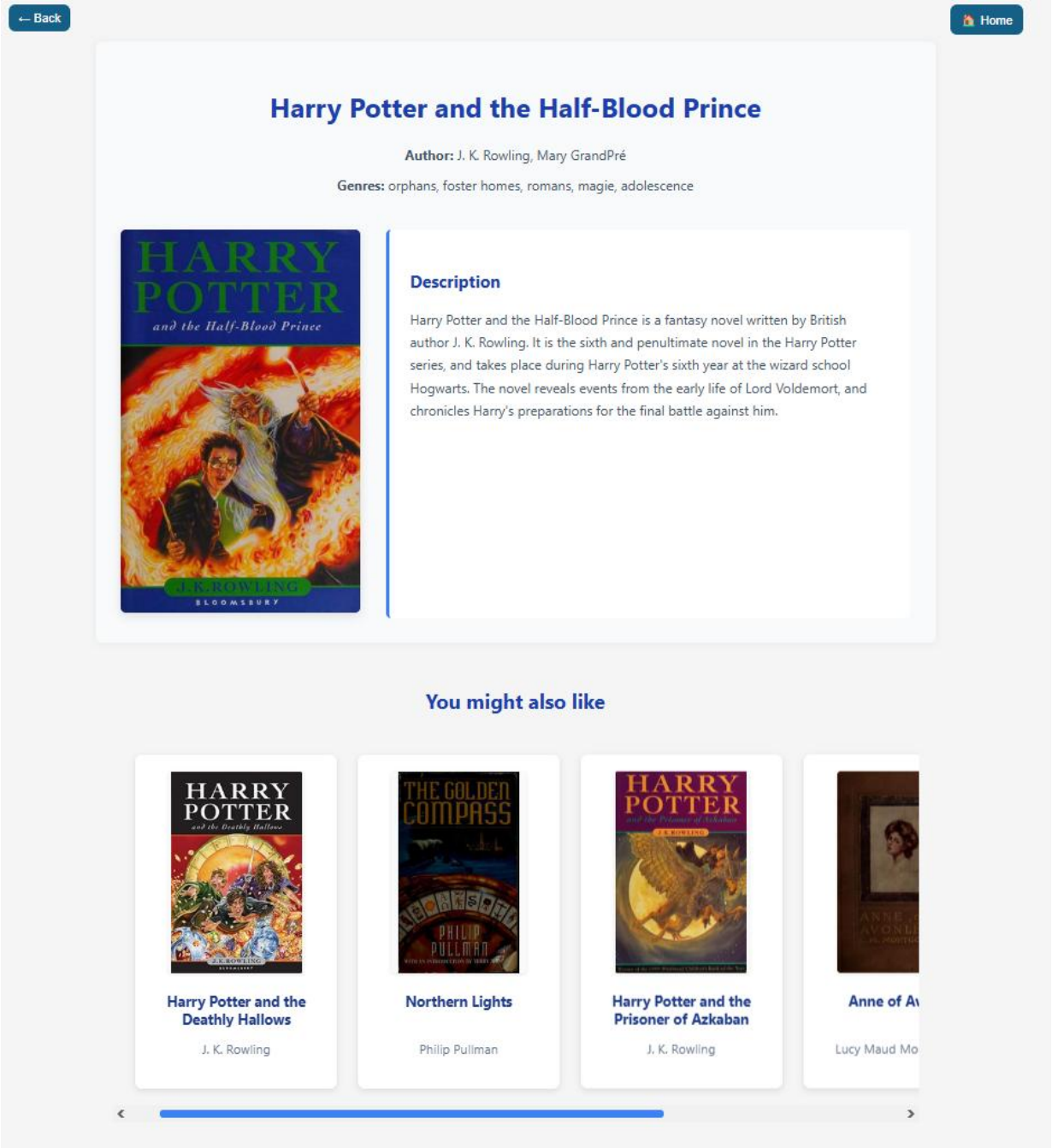



Рисунок 4.5.1 – Сторінка із детальною інформацією про книгу

Користувач має можливість повернутися на попередню сторінку, на головну сторінку, а також отримати детальну інформацію щодо будь-якої книги з нових рекомендацій внизу сторінки.

### 4.6 Перегляд детальної інформації про автора

Після отримання рекомендацій на основі автора користувач має можливість натиснути на картку будь-якого автора та отримати детальну інформацію про нього і топ його книг.

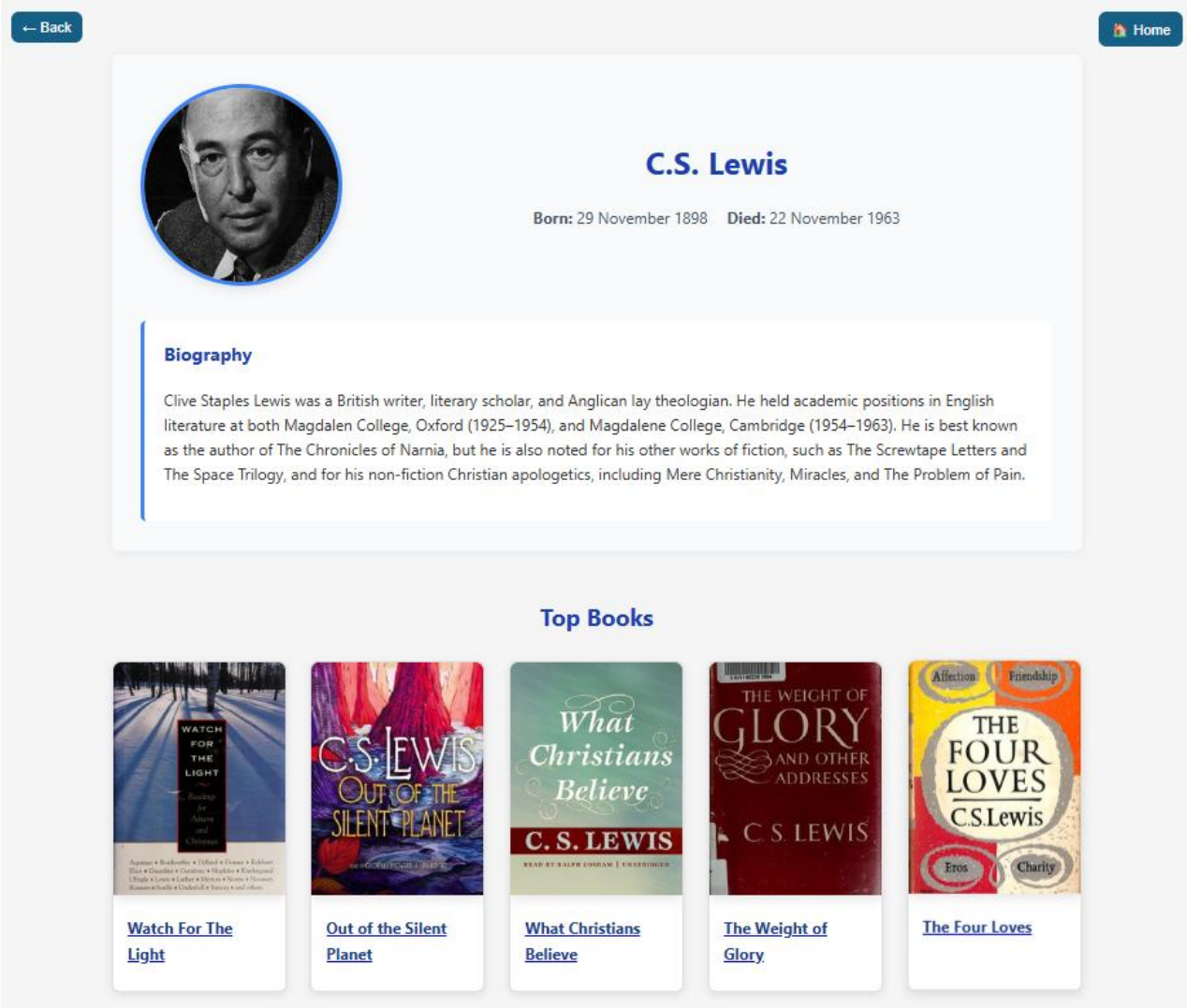



Рисунок 4.7.1 – Сторінка із детальною інформацією про автора

Користувач має можливість натиснути на будь-яку з найбільш популярних робіт автора та перейти на сторінку із детальною інформацією про цю книгу.

### 4.7 Обробка помилок

#### 4.7.1 Відсутність результатів

У випадку некоректного вводу імені автору/назви книги, або якщо обрано некоректну комбінацію жанрів, а також при відсутності результатів користувачу виводяться наступні повідомлення про помилку:

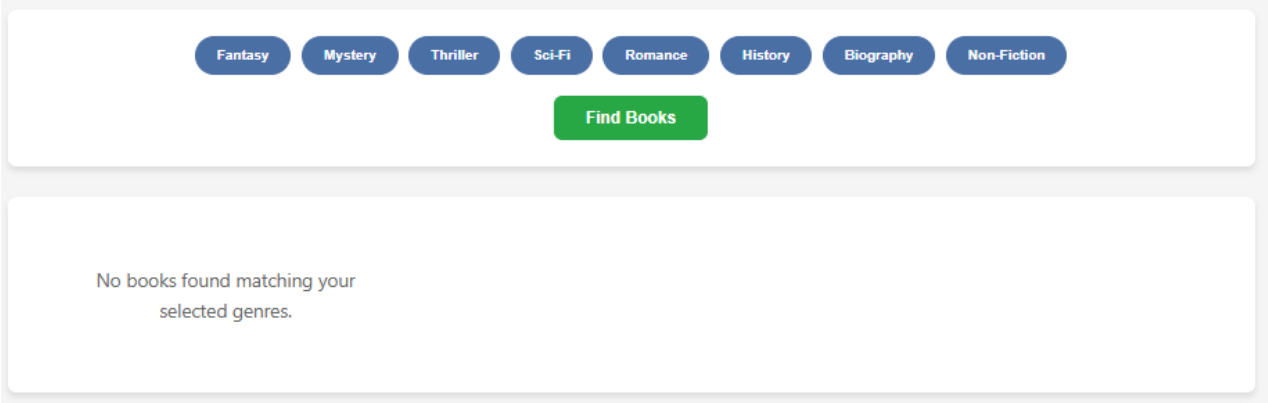



Рисунок 4.7.1.1 – Вивід повідомлення про відсутність результатів на сторінці для пошуку книг на основі жанрів

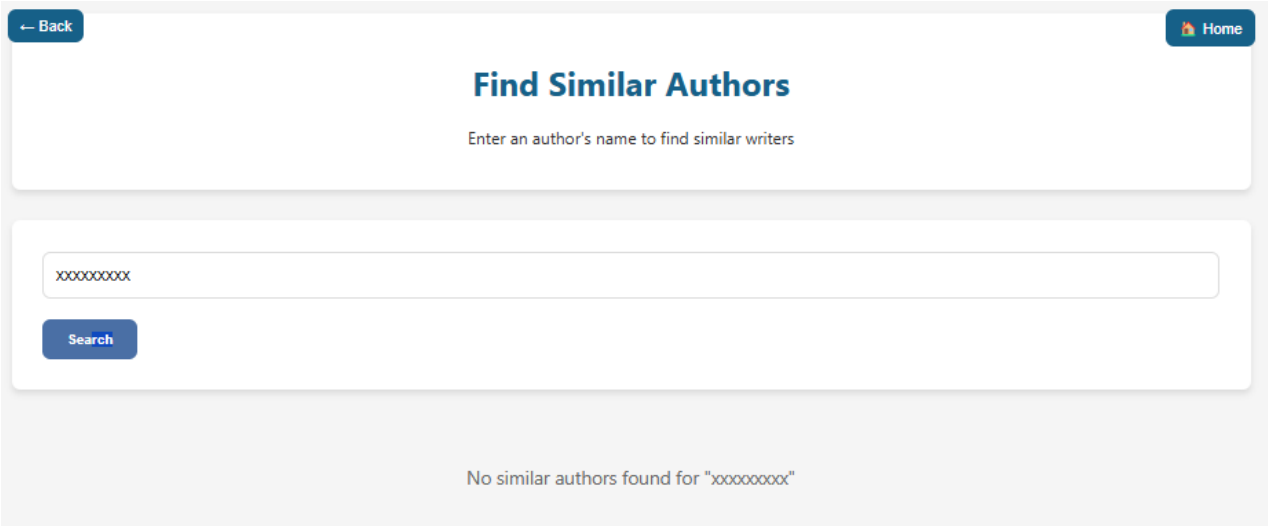



Рисунок 4.7.1.2 – Вивід повідомлення про відсутність результатів на сторінці для пошуку схожих авторів

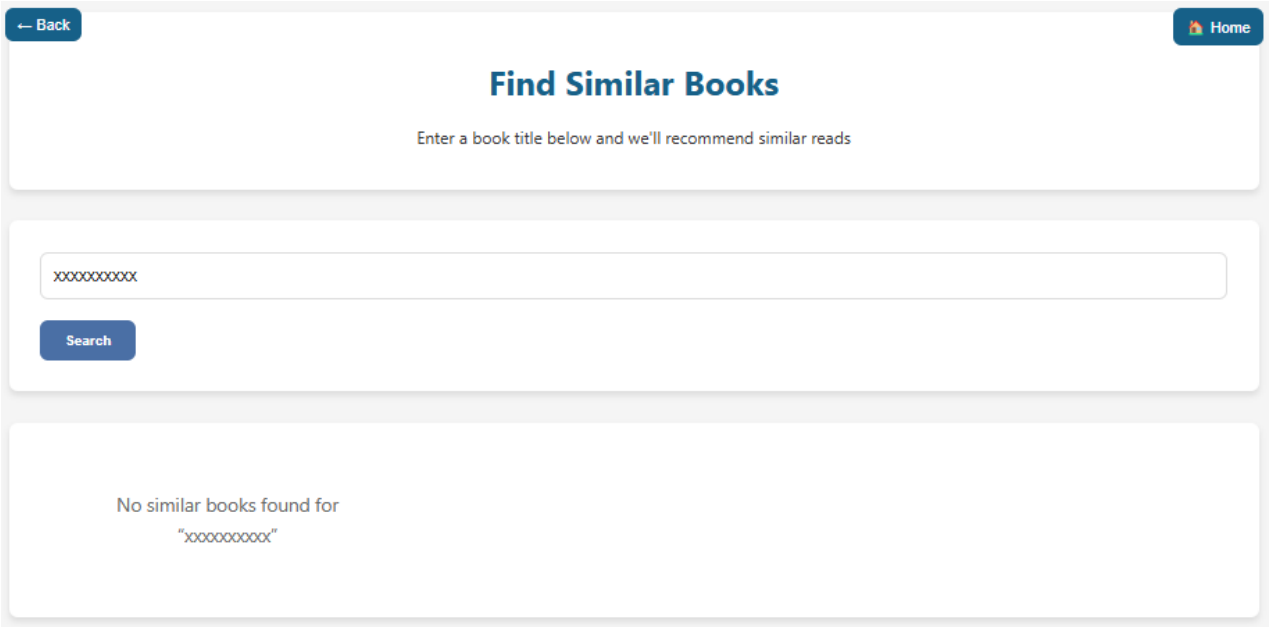



Рисунок 4.7.1.3 – Вивід повідомлення про відсутність результатів на сторінці для пошуку схожих книг

### 4.7.2 Відсутність додаткових результатів для підвантаження

На сторінці для пошуку на основі книги, у випадку якщо користувач багаторазово натискає на кнопку “Load More” для підвантаження додаткових результатів, а фактично результати вже закінчилися кнопка “Load More” стає недоступною і виводиться повідомлення про вичерпання ліміту результатів.

### 4.7.3 Відсутність зображення

У випадку відстутності зображення застостовуються стандартні зображення для обкладинки та портрету.

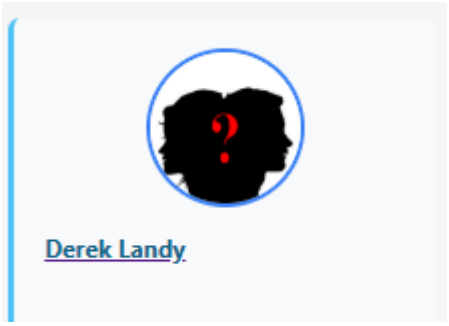



Рисунок 4.8.3.1 – Стандартне зображення для портрету автора


Рисунок 4.8.3.1 – Стандартне зображення для обкладинки книги

### 4.7.4 Відсутність опису книги або автора

У випадку відсутності опису для користувача виводиться стандартне повідомлення.

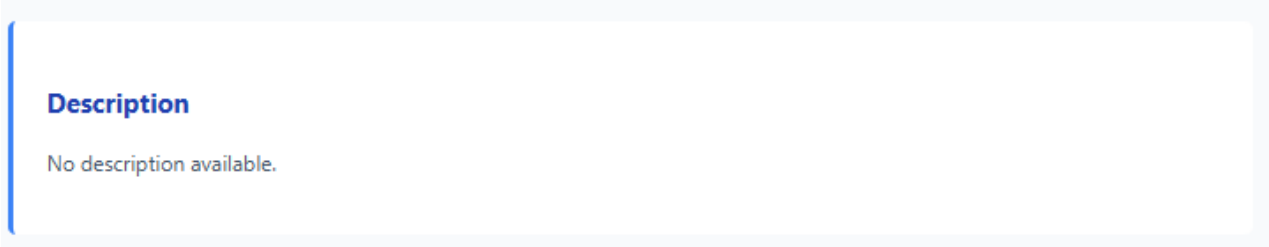



Рисунок 4.8.4.1 – Повідомлення про відсутність опису

## 4.8 Подальший розвиток веб-застосунку

Веб-застосунок і зараз забезечує доволі комфортний процес отримання рекомендацій, проте для поліпшення якості користувацького досвіду можна реалізувати декілька покращень:

- Підключення додаткового API – інформація у OpenLibrary API іноді буває неповною, за допомогою підключення додаткового API (наприклад Google Books API) можна забезпечити підвищення якості рекомендацій за допомогою добору інформації із двох і більше джерел, а також із більшою вірогідністю забезпечувати повноцінне заповнення

інформації про об’єкт.

- Додавання нових фільтрів для отримання рекомендацій – можна суттєво розширити функціонал застосунку та його гнучкість завдяки введеню нових фільтрів для пошуку книг, наприклад пошук за країною, за мовою оригіналу, за датою виходу, і т.п.
- Додавання облікових записів та персоналізації — користувачі зможуть створювати власні профілі, зберігати історію пошуків, оцінювати книги, створювати списки бажаного (wishlist) та переглянутого. Це дозволить реалізувати більш точні персоналізовані рекомендації.
- Впровадження гібридних рекомендацій — додавання фільмів, подкастів або статей, пов’язаних за темами, жанрами або авторами.

# ВИСНОВОК ДО РОЗДІЛУ 4

У цьому розділі проведено огляд та тестування розробленого веб-застосунку, розглянуто реакцію системи на помилки і нестачу даних, потенціальні шляхи покращення веб-застосунку.

Клієнтська частина забезпечує комфортний для користувача інтерфейс шляхом наявності та інтуїтивно зрозумілому розташуванню кнопок для навігації або відправки запиту, системи автодопомовнення для пошукового поля, якісному оформленню рекомендацій, можливості переглянути сторінку із детальною інформацією для будь-якої з них.

Забезпечено обробку найпопулярніших помилок, як-то ввід неіснуючого імені автора або назви книги, шляхом виведення напису із описом помилки. У випадках, коли частина інформації відсутня у API забезпечено заміну відсутніх даних на заготовлені або повідомлення користувача про нестачу даних. Наприклад, у випадку коли немає обкладинки книги або портрету автора у їх якості використовується заготовлене стандартне зображення, якщо немає опису – у полі для нього виводиться повідомлення про це, якщо немає рейтингу – виводиться відповідне повідомлення.

Розглянуто основні можливі напрямки для покращення та подальшого розвитку веб-застосунку.

Отже, розроблений веб-застосунок забезпечує приємний користувацький досвід та надає користувачу доступ до якісних та актуальних рекомендацій за фільтрами.

# ВИСНОВКИ

Результатом виконання бакалаврського проєкту є веб-застосунок для рекомендацій книг на основі користувацьких уподобань, який надає доступ до свого функціоналу без суворих вимог до користувацького пристрою, та має сучасний інтерфейс і власну систему створення рекомендацій.

У першому розділі дипломного проєкту було проведено аналіз предметної області, досліджено рекомендаційні системи та їх типи, переваги та недоліки, розглянуто найпопулярніші існуючі сервіси для рекомендацій книг, джерела інформації (API) для потенційного використання у проєкті, їх переваги та недоліки. Проведення аналізу допомогло обрати технології та засоби для подальшої розробки веб-застосунку, ознайомитися із основним функціоналом який має бути у таких сервісах та правильно обрати тип рекомендаційної системи для застосування у власному проєкті.

У другому розділі було проведено огляд технологій обраних для розробки веб-застосунку, в тому числі технологій для реалізацій серверної та клієнтської частини веб-застосунку, джерел даних (API), розглянуто історію їх виникнення, застосування у сучасних продуктах, переваги та недоліки їх використання для розробки веб-застосунків. Проведення дослідження та огляду технологій розробки допомогло в тому числі розглянути та уникнути проблем, які часто виникають під час розробки та створити правильну архітектуру для проєкту.

У третьому розділі розглянуто реалізацію веб-застосунку, детально описано основні програмні модулі та їх призначення, наведено конкретні функції та розглянуто їх використання у веб-застосунку. Проаналізовано основні принципи функціонування системи, пояснено її архітектуру, зосереджено увагу на маршрутизації запитів та створених маршрутах. Продемонстровано розроблену рекомендаційну систему із різним алгоритмом роботи для різних фільтрів, що дозволяє гнучко створювати рекомендації на основі користувацьких уподобань. Розглянуто взаємодію з API для отримання

даних. Метою третього розділу дипломного проєкту є аналіз та дослідження архітектури та основних програмних компонентів розробленої системи.

У четвертому розділі проведено огляд та тестування веб-застосунку, перевірено обробку помилок системою, отримано рекомендації за кожним з критерієв. Це дозволило отримати типовий користувацький досвід та оцінити інтерфейс, швидкодію та якість рекомендацій отриманих в результаті роботи веб-застосунку. Цей розділ дозволяє ознайомитися із фінальною версією розробленого продукту очима користувача, виправити існуючі помилки та підвести підсумки.

Під час виконання бакалаврського проєкту було проведено аналіз предметної області, оглянуто технології для розробки та програмну реалізацію веб-застосунку, проведено тестування його роботи. Результатом розробки є веб-застосунок із сучасним інтерфейсом, який заохочує людей до читання у епоху постійного цифрового шуму та дозволяє відкрити нових авторів та книги відповідно до користувацьких уподобань.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. What are Recommender Systems? [Електронний ресурс] – Режим доступу до ресурсу: https://coralogix.com/ai-blog/what-are-recommender-systems-use-cases-types-and-techniques/
2. What is content-based filtering? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/think/topics/content-based-filtering
3. Pros and Cons of Collaborative Filtering [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@ashmi_banerjee/pros-and-cons-of-collaborative-filtering-9c3aa4ce44f6
4. Advantages And Disadvantages Of Hybrid Use Cases Of Filtering [Електронний ресурс] – Режим доступу до ресурсу: https://www.slidegeeks.com/advantages-and-disadvantages-of-hybrid-use-cases-of-filtering-methods-demonstration-pdf
5. Knowledge-Based Recommenders [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/%40paul0/knowledge-based-rs-b4547b84108a
6. What is an API? [Електронний ресурс] – Режим доступу до ресурсу: https://konghq.com/blog/learning-center/what-is-api
7. Open Library API [Електронний ресурс] – Режим доступу до ресурсу: https://openlibrary.org/developers/api
8. Google Books APIs [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/books
9. Goodreads API [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/%40vaswhite11/goodreads-api-2022-deef9af61e88
10. BookBrainz user guide [Електронний ресурс] – Режим доступу до ресурсу: https://bookbrainz-user-guide.readthedocs.io/en/latest/#home/

11. Goodreads guide [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Goodreads
12. The StoryGraph [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/The_StoryGraph
13. Whichbook [Електронний ресурс] – Режим доступу до ресурсу: https://www.whichbook.net/about/our-story
14. Python History [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/History_of_Python
15. Python Pros and Cons [Електронний ресурс] – Режим доступу до ресурсу: https://www.netguru.com/blog/python-pros-and-cons
16. JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/JavaScript
17. FastAPI History, Design and Future – Режим доступу до ресурсу: https://fastapi.tiangolo.com/history-design-future/
18. HTML Styles CSS [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3schools.com/html/html_css.asp
19. Using FastAPI to Build Python Web APIs [Електронний ресурс] – Режим доступу до ресурсу: https://realpython.com/fastapi-python-web-apis/
20. Building a Website with Python, FastAPI, and Streamlit [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@obaff/building-a-website-with-python-fastapi-and-streamlit-418f48c41af2
21. Python & APIs: A Winning Combo for Reading Public Data [Електронний ресурс] – Режим доступу до ресурсу: https://realpython.com/python-api/
22. How to Use an API in Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.dataquest.io/blog/api-in-python/
23. Recommender system [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Recommender_system

24. Content-based Recommendation System [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@prateekgaurav/step-by-step-content-based-recommendation-system-823bbfd0541c
25. JavaScript Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3schools.com/js/DEFAULT.asp
26. Recommendation System in Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/recommendation-system-in-python/
27. Python Web Development [Електронний ресурс] – Режим доступу до ресурсу: https://www.browserstack.com/guide/web-development-in-python-guide
28. FastAPI Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://fastapi.tiangolo.com/tutorial/

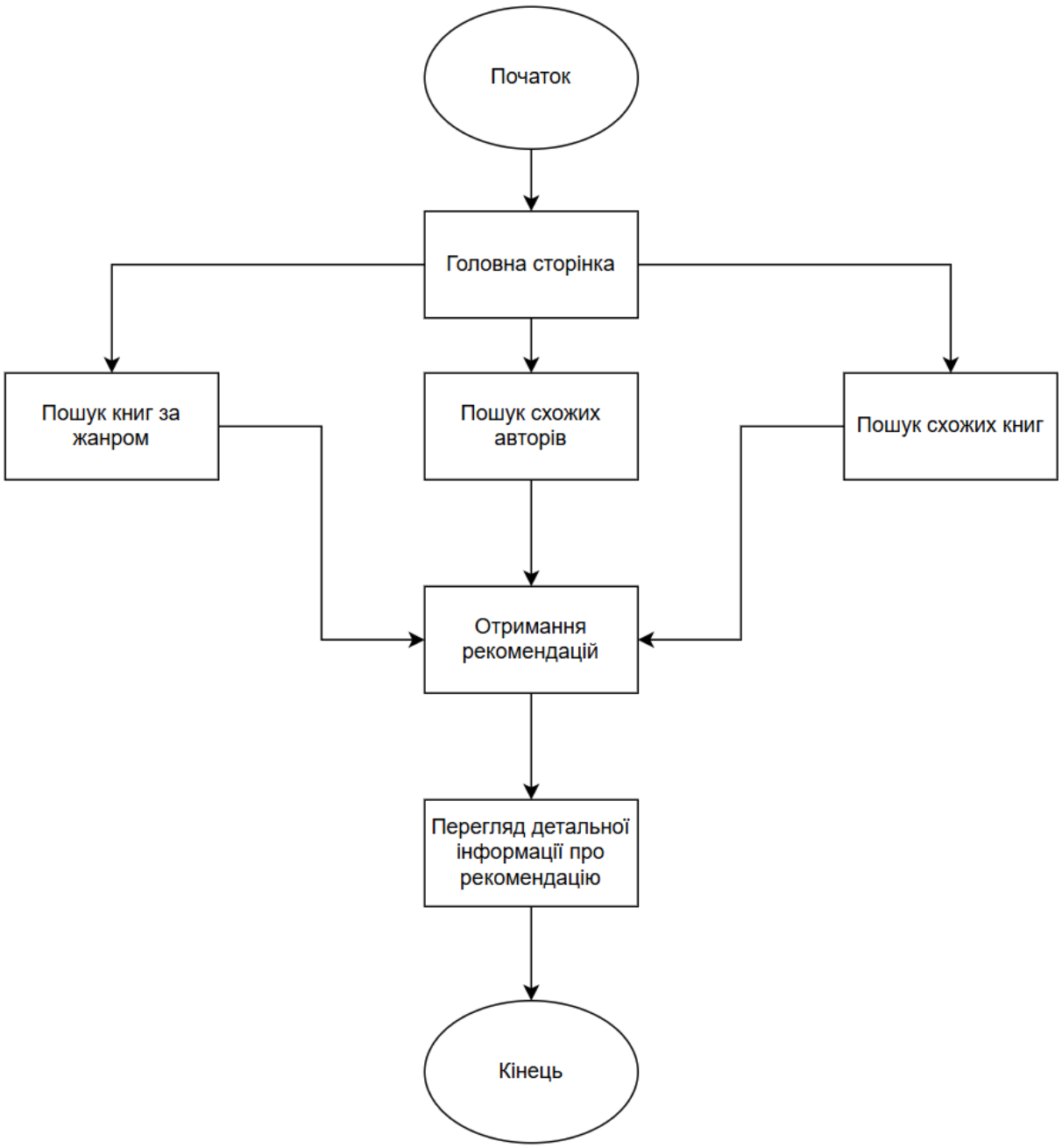
# ДОДАТОК 1

Веб-застосунок для рекомендацій книг на основі користувацьких уподобань

Схема роботи веб-застосунку

ІАЛЦ.467200.007 Д1

Київ – 2025




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.007 Д1 | | | |
| Зм. | Арк. | № докум. | Підпис | Дата | | | | |
| Розробив | | Гадієв Р.А. | | | **Веб-застосунок для рекомендацій кнг на основі користувацьких уподобань** | Літ. | Аркуш | Аркушів |
| Перевірив | | Волокита А.М. | | | | | 1 | |
| Реценз. | | Орленко С.П. | | | Схема роботи веб-застосунку | | | |
| Н. Контр. | | | | | | | | |
| Затвердив | | | | | | | | |

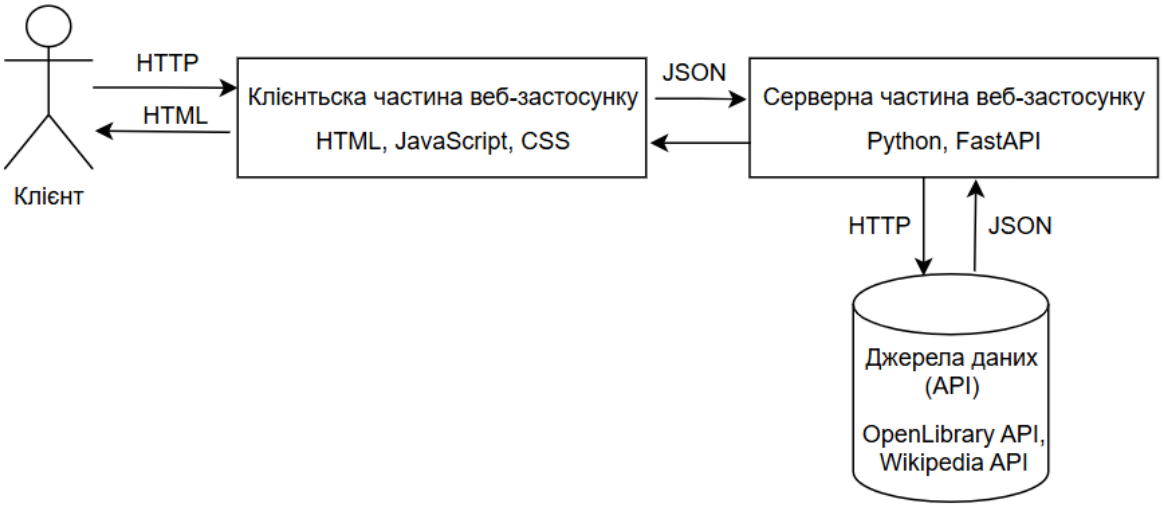
# ДОДАТОК 2

Веб-застосунок для рекомендацій книг на основі користувацьких уподобань

Структурна схема архітектури веб-застосунку

ІАЛЦ.467200.007 Д2

Київ – 2025




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.007 Д2 | | | |
| Зм. | Арк. | № докум. | Підпис | Дата | | | | |
| Розробив | | Гадієв Р.А. | | | **Веб-застосунок для рекомендацій кнг на основі користувацьких уподобань** | Літ. | Аркуш | Аркушів |
| Перевірив | | Волокита А.М. | | | | | 1 | |
| Реценз. | | Орленко С.П. | | | Структурна схема архітектури веб-застосунку | | | |
| Н. Контр. | | | | | | | | |
| Затвердив | | | | | | | | |

# ДОДАТОК 3

Веб-застосунок для рекомендацій книг на

Основі користувацьких уподобань

Схема маршрутизації запитів у веб-застосунку

ІАЛЦ.467200.007 Д3

Київ – 2025


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.007 Д3 | | |
| Зм. | Арк. | № докум. | Підпис | Дата | | | |
| Розробив | | Гадієв Р.А. | | | **Веб-застосунок для рекомендацій кнг на основі користувацьких уподобань** | Літ. | Аркуш 1 / Аркушів 1 |
| Перевірив | | Волокита А.М. | | | | | |
| Реценз. | | Орленко С.П. | | | Схема маршрутизації запитів у веб-застосунку | | |
| Н. Контр. | | | | | | | |
| Затвердив | | | | | | | |

# ДОДАТОК 4

Веб-застосунок для рекомендацій книг на основі користувацьких уподобань

Лістинг програмного коду

ІАЛЦ.467200.007 Д4

Аркушів - 55

Київ - 2025

## main.py:

```
import logging
from logging.config import dictConfig
from pathlib import Path
import uvicorn
from fastapi import FastAPI
from starlette.responses import FileResponse
from starlette.staticfiles import StaticFiles
from backend.routers.filters import router as filters_router
from backend.routers.user import router as users_router

# Logging configuration
LOG_CONFIG = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "default": {
            "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
        }
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "formatter": "default",
            "stream": "ext://sys.stdout"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "default",
            "filename": "app.log",
            "mode": "a"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": ["console", "file"]
    }
```

```
}

dictConfig(LOG_CONFIG)
        logger = logging.getLogger(__name__)

            app = FastAPI()

            # Absolute path to project root
            BASE_DIR = Path(__file__).resolve().parent.parent

# Mount static files directory
app.mount(
    "/static",
    StaticFiles(directory=BASE_DIR / "frontend" / "static"),
    name="static"
)

# Serve main index.html
@app.get("/")
def main():
    return FileResponse(BASE_DIR / "frontend" / "templates" / "index.html")

# Register routers
app.include_router(users_router)
app.include_router(filters_router)

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

## models.py:

```
from typing import List, Optional, Dict, Any
from pydantic import BaseModel

class Book(BaseModel):
    key: str
    title: str
    authors: List[str] = []
```

```
    genres: List[str] = []
    rating: Optional[float] = None
    rating_count: Optional[int] = None
    cover_id: Optional[int] = None
    publish_year: Optional[int] = None

    def dict(self, **kwargs):
        data = super().dict(**kwargs)
        return data

class Author(BaseModel):
    key: str
    name: str
    birth_date: Optional[str] = None
    death_date: Optional[str] = None
    bio: Optional[str] = None
    works_count: Optional[int] = None
    subjects: List[str] = []
    links: List[Dict[str, Any]] = []
    top_work: Optional[str] = None
    alternate_names: List[str] = []
    rating: Optional[float] = None
    similarity_score: Optional[float] = None
    photo_id: Optional[int] = None

class BookDetailSchema(BaseModel):
    detail: dict
    recommendations: List[Book]
```

## config.py:

```
# ---------------------------------------------------------------------------------------------------------------
#                                        Genre-based parameters
# ---------------------------------------------------------------------------------------------------------------

BOOKS_MIN_RATING_DEFAULT: float   = 3.5
BOOKS_MIN_REVIEWS_DEFAULT: int    = 10
BOOKS_PAGE_LIMIT_DEFAULT: int     = 20
BOOKS_OFFSET_DEFAULT: int         = 0
```

```
# ------------------------------------------------------------------------------------------------------------
#                                        Author-based parameters
# ------------------------------------------------------------------------------------------------------------

AUTHOR_LIMIT_DEFAULT: int = 20


# ------------------------------------------------------------------------------------------------------------
#                                        Book-based parameters
# ------------------------------------------------------------------------------------------------------------

BOOK_PAGE_LIMIT_DEFAULT: int = 20
```

## open_library_api.py:

```
import math
from typing import List, Dict, Optional, Union, Set, Literal
import requests
import logging
from collections import Counter
from itertools import combinations, islice
from functools import lru_cache
from concurrent.futures import ThreadPoolExecutor, as_completed
from urllib.parse import quote
import re
from backend.models import Author
logger = logging.getLogger(__name__)

from backend.config import (
    BOOKS_MIN_REVIEWS_DEFAULT,
    BOOKS_PAGE_LIMIT_DEFAULT,
    BOOKS_OFFSET_DEFAULT,
    AUTHOR_LIMIT_DEFAULT
)

# Global HTTP session for connection pooling
diploma_session = requests.Session()
```

```
diploma_session.headers.update({'User-Agent': 'DiplomaProject/1.0'})

# API endpoints
OL_SEARCH_URL = 'https://openlibrary.org/search.json'
OL_AUTHORS_URL = 'https://openlibrary.org/search/authors.json'
OL_AUTHOR_WORKS_URL = 'https://openlibrary.org/authors/{author_key}/works.json'
OL_AUTHOR_DETAILS_URL = 'https://openlibrary.org/authors/{author_key}.json'
OL_SUBJECT_URL = 'https://openlibrary.org/subjects/{subject}.json'



# ------------------------------------------------------------------------------------------------------------
#                                        General
# ------------------------------------------------------------------------------------------------------------

def get_book_details(work_key: str) -> Dict:
    """
    Retrieve detailed information about a book using its work key.

    This function fetches metadata for a book from the OpenLibrary API, including title,
    subjects, authors, and cover images. Additionally, it queries the Wikipedia REST API
    to extract a short description of the book based on its title.

    Args:
        work_key (str): The unique work identifier of the book (e.g., 'OL12345W').

    Returns:
        Dict: A dictionary containing book metadata including 'key', 'title', 'description',
            'subjects', 'authors', and 'covers'.
    """
    url  = f"https://openlibrary.org/works/{work_key}.json"
    resp = diploma_session.get(url, timeout=10)
    resp.raise_for_status()
    data = resp.json()

    authors = []
    for a in data.get("authors", []):
        auth_key = a.get("author", {}).get("key", "").split("/")[-1]
        det = get_author_details(auth_key)
        if det:
            authors.append(det.name)
```

```
    raw_wiki    = fetch_wikipedia_summary(data.get("title", ""))
    description = raw_wiki.strip() if raw_wiki else None

    return {
        "key":         work_key,
        "title":       data.get("title"),
        "description": description,
        "subjects":    data.get("subjects", []),
        "authors":     authors,
        "covers":      data.get("covers", []),
    }


def fetch_wikipedia_summary(name: str) -> Optional[str]:
    """
    Fetch a short summary description of a book or author from the Wikipedia REST API.

    This function takes a name (title or author), formats it for a Wikipedia URL,
    and retrieves the corresponding summary text from Wikipedia's REST API.

    Args:
        name (str): The title of the book or name of the author to search for.

    Returns:
        Optional[str]: A short summary text extracted from Wikipedia, or None if the request fails.
    """
    title = name.replace(' ', '_')
    url   = f'https://en.wikipedia.org/api/rest_v1/page/summary/{title}'
    try:
        r = requests.get(url, timeout=10)
        r.raise_for_status()
        return r.json().get('extract')
    except Exception:
        return None


# ----------------------------------------------------------------------------------------------------------
#                                        Genre-based
# ----------------------------------------------------------------------------------------------------------

def search_books_ol(
```

```
    genres: Union[str, List[str]],
    min_reviews: int = BOOKS_MIN_REVIEWS_DEFAULT,
    limit: int = BOOKS_PAGE_LIMIT_DEFAULT,
    offset: int = BOOKS_OFFSET_DEFAULT,
) -> List[Dict]:
    """
    Search for books in the OpenLibrary API by genre and minimum number of reviews.

    This function constructs a subject-based query using one or more genre keywords
    and fetches a list of books that have at least the specified number of user ratings.
    Results are filtered and paginated using the given parameters.

    Args:
        genres (Union[str, List[str]]): One or more genre keywords to include in the subject query.
        min_reviews (int): Minimum number of user reviews required to include a book.
        limit (int): Maximum number of results to return per request.
        offset (int): Pagination offset for the search results.

    Returns:
        List[Dict]: A list of raw book data dictionaries returned by the OpenLibrary API.
    """
    if isinstance(genres, str):
        genre_list = [genres]
    else:
        genre_list = genres

    subject_query = ' AND '.join(f'subject:{g}' for g in genre_list)
    params = {
        'q': subject_query,
        'ratings_count': f'[{min_reviews} TO *]',
        'limit': limit,
        'offset': offset,
        'fields': ','.join([
            'key','title','subtitle','author_name',
            'first_publish_year','edition_count','publisher',
            'language','subject','cover_i','ratings_count',
            'ratings_average','isbn'
        ]),
    }

    try:
```

```
        resp = requests.get(OL_SEARCH_URL, params=params, timeout=15)
        resp.raise_for_status()
        return resp.json().get('docs', [])
    except Exception as e:
        logger.error(f"Error searching books: {e}")
        return []


# ------------------------------------------------------------------------------------------------------------
#                                        Author-based
# ------------------------------------------------------------------------------------------------------------

@lru_cache(maxsize=128)
def get_subjects_from_works(author_key: str) -> List[str]:
    """
    Fetch the most frequent subjects from an author's most popular works using the OpenLibrary API.

    This function retrieves up to 50 works by the specified author, sorts them by edition count
    to estimate popularity, and analyzes the top 20 works to extract their subjects.
    It returns the 5 most common subjects across those works. Results are cached for efficiency.

    Args:
        author_key (str): The unique key of the author (e.g., 'OL123A').

    Returns:
        List[str]: A list of the top 5 most frequent subject keywords from the author's works.
    """
    try:
        url = OL_AUTHOR_WORKS_URL.format(author_key=author_key)
        params = {'limit': 50, 'fields': 'subjects,edition_count'}
        data = diploma_session.get(url, params=params, timeout=10).json()
        entries = data.get('entries', [])
        entries.sort(key=lambda x: x.get('edition_count', 0), reverse=True)
        subjects = [s.lower()
                    for work in entries[:20]
                    for s in work.get('subjects', [])
                    if isinstance(s, str)]
        return [subj for subj, _ in Counter(subjects).most_common(5)]
    except Exception as e:
        logger.error(f'Error fetching works subjects for {author_key}: {e}')
        return []
```

```
@lru_cache(maxsize=128)
def get_author_details(author_key: str) -> Optional[Author]:
    """
    Retrieve detailed metadata for an author using the OpenLibrary API and Wikipedia.

    This function fetches author information such as name, birth/death dates, subjects,
    biography (from Wikipedia), and other metadata using the author's unique key.
    The result is cached for performance optimization.

    Args:
        author_key (str): The unique key or full URI of the author (e.g., 'OL123A' or '/authors/OL123A').

    Returns:
        Optional[Author]: An Author object with detailed information if successful; otherwise, None.
    """
    try:
        key  = author_key.split('/')[-1]
        url  = OL_AUTHOR_DETAILS_URL.format(author_key=key)
        data = diploma_session.get(url, timeout=15).json()
        raw_wiki = fetch_wikipedia_summary(data.get('name', ''))
        bio = raw_wiki.strip() if raw_wiki else None
        photos   = data.get('photos', [])
        photo_id = photos[0] if photos else None

        return Author(
            key=key,
            name=data.get('name', 'Unknown Author'),
            birth_date=data.get('birth_date'),
            death_date=data.get('death_date'),
            bio=bio,
            works_count=data.get('works_count'),
            subjects=data.get('top_subjects', [])[:5],
            links=data.get('links', []),
            top_work=data.get('top_work'),
            alternate_names=data.get('alternate_names', []),
            rating=data.get('ratings_average'),
            photo_id=photo_id
        )
    except Exception as e:
```

```
        logger.error(f'Error fetching author details for {author_key}: {e}')
        return None



def find_similar_authors(target_author: str, limit: AUTHOR_LIMIT_DEFAULT) -> List[Dict]:
    """
    Find authors similar to a target author based on shared subject combinations.

    This function extracts the top subjects associated with the target author and generates
    all possible subject pairs. For each combination, it performs an AND-query search via
    the OpenLibrary API to retrieve authors whose works match those subjects. Authors are scored
    by how frequently they appear across all combinations, and the top results are returned.

    Args:
        target_author (str): The name of the target author for whom similar authors are to be found.
        limit (int): Maximum number of similar authors to return.

    Returns:
        List[Dict]: A list of author dictionaries containing 'name', 'key', and 'score',
                    sorted by descending relevance.
    """
    try:
        params = {'q': target_author, 'limit': 1}
        search_data = diploma_session.get(OL_AUTHORS_URL, params=params, timeout=10).json()
        docs = search_data.get('docs', [])
        if not docs:
            return []
        primary = docs[0]
        primary_key = primary.get('key', '').split('/')[-1]

        target_subs = [s.lower() for s in primary.get('top_subjects', [])]
        if not target_subs:
            target_subs = get_subjects_from_works(primary_key)
        if not target_subs:
            target_subs = ['science']
        logger.debug(f"Target subjects for {target_author}: {target_subs}")

        combos = list(combinations(target_subs, 2))
        if not combos:
            combos = [(s,) for s in target_subs]
```

```
def fetch_combo_authors(combo):
    """
    Fetch authors whose works match a specific combination of subjects.

    This function constructs an AND-based subject query and sends a request to the OpenLibrary API
    to find authors who have written works that match all subjects in the given combination.

    Args:
        combo (Tuple[str, ...]): A tuple containing one or more subject keywords.

    Returns:
        List[Dict]: A list of raw author data dictionaries matching the subject combination.
    """
    query = ' AND '.join(f'subject:"{s}"' for s in combo)
    url = OL_SEARCH_URL
    params = {
        'q': query,
        'limit': 50,
        'fields': 'author_key,author_name'
    }
    return diploma_session.get(url, params=params, timeout=10).json().get('docs', [])

author_scores = Counter()
author_names: Dict[str, str] = {}
with ThreadPoolExecutor(max_workers=8) as executor:
    futures = {executor.submit(fetch_combo_authors, combo): combo for combo in combos}
    for future in as_completed(futures):
        try:
            docs = future.result()
            for doc in docs:
                keys = doc.get('author_key') or []
                names = doc.get('author_name') or []
                if not keys or not names:
                    continue
                key = keys[0].split('/')[-1]
                if key.lower() == primary_key.lower():
                    continue
                author_scores[key] += 1
                author_names[key] = names[0]
        except Exception as e:
```

```
                logger.error(f"Error fetching combo result: {e}")

        results = []
        for key, score in author_scores.most_common(limit):
            results.append({'name': author_names.get(key), 'key': key, 'score': score})
        return results

    except Exception as e:
        logger.error(f'Error finding similar authors: {e}')
        return []


# ------------------------------------------------------------------------------------------------------------
#                                        Book-based
# ------------------------------------------------------------------------------------------------------------

def find_similar_books(
    target_book: str,
    limit: int = 10,
    offset: int = 0,
    max_subjects: int = 10
) -> List[Dict]:
    """
    Find books similar to a given title based on shared subjects from OpenLibrary.

    This function first retrieves the target book and extracts its top subjects.
    It then searches for books that share those subjects using OR-based queries and ranks
    them by subject overlap. About 70% of the returned results are required to share at least
    70% of the target's subjects (strict match), and the remaining 30% may share fewer subjects
    (exploratory match). Books with the same title or key as the original are excluded.

    Args:
        target_book (str): Title of the book to base the similarity search on.
        limit (int): Maximum number of similar books to return.
        offset (int): Offset for paginated API requests.
        max_subjects (int): Maximum number of subjects to consider from the target book.

    Returns:
        List[Dict]: A list of dictionaries representing similar books,
                    each including a 'ratio' field indicating subject overlap.
    """
```

```
try:
    if offset == 0:
        resp = diploma_session.get(
            OL_SEARCH_URL,
            params={'q': target_book, 'limit': 1, 'fields': 'key,title,subject'},
            timeout=10
        )
        if not resp.ok:
            logger.error(f"Lookup error for '{target_book}': {resp.status_code}")
            return []
        docs = resp.json().get('docs', [])
        if not docs:
            return []

        target = docs[0]
        find_similar_books._target_cache = target
    else:
        target = getattr(find_similar_books, "_target_cache", None)
        if target is None:
            logger.error("Offset > 0 but target book metadata not cached.")
            return []

    target_key = target.get('key')
    user_query = target_book.lower().strip()

    raw_subjects = [s for s in target.get('subject', []) if isinstance(s, str)]
    if not raw_subjects:
        return []

    top_subjects = raw_subjects[:max_subjects]
    target_set = {s.lower() for s in top_subjects}

    or_clause = " OR ".join(f'subject:"{s}"' for s in top_subjects)
    pool_resp = diploma_session.get(
        OL_SEARCH_URL,
        params={
            'q': or_clause,
            'limit': limit * 10,
            'offset': offset,
            'fields': 'key,title,author_name,subject,cover_i,ratings_count,ratings_average'
        },
```

```
    timeout=15
)
if not pool_resp.ok:
    logger.error(f"Search pool error: {pool_resp.status_code} for query {or_clause}")
    return []
pool = pool_resp.json().get('docs', [])

candidates = []
seen = set()
for doc in pool:
    key = doc.get('key')
    title = (doc.get('title') or "").lower()

    if not key or key == target_key or key in seen or (user_query and user_query in title):
        continue
    seen.add(key)

    cand_subjects = {s.lower() for s in doc.get('subject', []) if isinstance(s, str)}
    ratio = len(target_set & cand_subjects) / len(target_set) if target_set else 0.0
    candidates.append({**doc, 'ratio': ratio})

strict = [c for c in candidates if c['ratio'] >= 0.7]
explore = [c for c in candidates if 0 < c['ratio'] < 0.7]

strict_n = math.ceil(limit * 0.7)
explore_n = limit - strict_n

strict.sort(key=lambda x: x['ratio'], reverse=True)
explore.sort(key=lambda x: x['ratio'], reverse=True)

selected = strict[:strict_n]
if len(selected) < strict_n:
    explore_n += (strict_n - len(selected))
selected += explore[:explore_n]

if len(selected) < limit:
    extras = strict[strict_n:] + explore[explore_n:]
    selected += extras[: limit - len(selected)]

return selected[:limit]
```

```
    except Exception as e:
        logger.error(f"Error finding similar books for '{target_book}': {e}", exc_info=True)
        return []
```

## recommender.py:

```
# ------------------------------------------------------------------------------------------------------------
#                                        Content-based recommender
# ------------------------------------------------------------------------------------------------------------
import math
from typing import List, Dict, Set
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from backend.api.open_library_api import (search_books_ol, get_author_details, find_similar_authors,
find_similar_books)
from backend.models import Book, Author
import logging
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
logger = logging.getLogger(__name__)
from backend.config import (
    BOOKS_MIN_RATING_DEFAULT,
    BOOKS_MIN_REVIEWS_DEFAULT,
    BOOKS_PAGE_LIMIT_DEFAULT,
    BOOKS_OFFSET_DEFAULT,
    AUTHOR_LIMIT_DEFAULT,
    BOOK_PAGE_LIMIT_DEFAULT
)


# ------------------------------------------------------------------------------------------------------------
#                                        Genre-based
# ------------------------------------------------------------------------------------------------------------

def recommend_by_genre(
    genres: List[str],
    min_rating: float = BOOKS_MIN_RATING_DEFAULT,
    min_reviews: int = BOOKS_MIN_REVIEWS_DEFAULT,
    limit: int = BOOKS_PAGE_LIMIT_DEFAULT,
```

```
    offset: int = BOOKS_OFFSET_DEFAULT
) -> List[Book]:
    """
    Recommend books based on genre filters, average rating, and number of reviews.

    This function retrieves books from OpenLibrary matching the specified genres,
    then filters them based on minimum rating and review count. It continues querying
    in batches until the required number of books is collected or no more are found.
    Results are sorted in descending order by rating.

    Args:
        genres (List[str]): List of genres (subjects) to filter by.
        min_rating (float): Minimum average rating required for a book to be included.
        min_reviews (int): Minimum number of reviews required.
        limit (int): Maximum number of books to return.
        offset (int): Starting point for pagination in the search results.

    Returns:
        List[Book]: A list of Book objects that match the filtering criteria,
                sorted by rating in descending order.
    """

    collected: List[Book] = []
    current_offset = offset

    while len(collected) < limit:
        raw_batch = search_books_ol(
            genres=genres,
            min_reviews=min_reviews,
            limit=limit,
            offset=current_offset
        )
        if not raw_batch:
            break

        for item in raw_batch:
            avg = item.get("ratings_average") or 0
            if avg >= min_rating:
                collected.append(Book(
                    key=item["key"],
                    title=item.get("title", "Unknown"),
```

```
                authors=item.get("author_name", []),
                cover_id=item.get("cover_i"),
                rating=avg,
                rating_count=item.get("ratings_count", 0),
                genres=item.get("subject", []),
                publish_year=item.get("first_publish_year")
            ))
            if len(collected) == limit:
                break

        if len(collected) < limit:
            current_offset += len(raw_batch)

    collected.sort(key=lambda x: x.rating or 0, reverse=True)
    return collected[:limit]



# ------------------------------------------------------------------------------------------------------------
#                                        Author-based
# ------------------------------------------------------------------------------------------------------------

def calculate_similarity(target_subjects: Set[str], candidate_subjects: List[str]) -> float:
    """
    Calculate the similarity score between two authors based on overlapping subjects.

    This function compares the set of subjects from a target author with the subjects
    of a candidate author and returns a normalized similarity score based on the proportion
    of shared subjects. If the target author has no subjects, a default base score is returned.

    Args:
        target_subjects (Set[str]): A set of subject tags associated with the target author.
        candidate_subjects (List[str]): A list of subject tags from the candidate author.

    Returns:
        float: A similarity score between 0.0 and 1.0 representing thematic overlap.
               Returns 0.4 if the target has no subjects.
    """
    if not target_subjects:
        return 0.4  # Default base score

    candidate_set = set(candidate_subjects)
```

```
    intersection = target_subjects & candidate_set

    base_score = min(1.0, len(intersection) / len(target_subjects))

    return min(1.0, base_score)


def recommend_similar_authors(target_author: str, limit: AUTHOR_LIMIT_DEFAULT) -> List[Author]:
    """
    Recommend authors similar to the given target author based on subject overlap.

    This function first retrieves a set of candidate authors that are thematically related
    to the target author (via shared subject tags). It then calculates a similarity score
    for each candidate using subject intersection and ranks them accordingly.
    Only the top 'limit' most similar authors are returned.

    Steps:
    1. Fetch a set of initial candidates (2 * limit) using subject-based lookup.
    2. Retrieve detailed information for the target author (including subjects).
    3. In parallel, retrieve full data for all candidates.
    4. For each candidate, compute a similarity score based on overlapping subjects.
    5. Sort candidates by score and return the top N results.

    Args:
        target_author (str): Name of the author for whom similar authors are to be recommended.
        limit (int): Number of similar authors to return.

    Returns:
        List[Author]: List of Author objects, each with an assigned similarity_score.
    """
    start = time.perf_counter()
    logger.info(f"Starting recommendation for author: {target_author}")

    candidates = find_similar_authors(target_author, limit=limit * 2)
    logger.debug(f"Initial candidates count: {len(candidates)}")
    if not candidates:
        return []

    target_data = get_author_details(candidates[0]['key'])
    if not target_data:
        return []
```

```
    target_subjects = set(s.lower() for s in target_data.subjects)

    results: List[Author] = []
    with ThreadPoolExecutor(max_workers=8) as executor:
        future_to_key = {executor.submit(get_author_details, c['key']): c['key'] for c in candidates}
        for future in as_completed(future_to_key):
            author = future.result()
            if not author:
                continue
            score = calculate_similarity(target_subjects, [s.lower() for s in author.subjects])
            author.similarity_score = score
            results.append(author)

    duration = time.perf_counter() - start
    logger.info(f"recommend_similar_authors took {duration:.2f}s")
    return sorted(results, key=lambda x: x.similarity_score, reverse=True)[:limit]




# ------------------------------------------------------------------------------------------------------------
#                                        Book-based
# ------------------------------------------------------------------------------------------------------------

def recommend_similar_books(
    target_book: str,
    limit: int = 10
) -> List[Dict]:
    """
    Recommend books similar to the given target title based on textual similarity.

    This function finds a list of candidate books using subject overlap with the target book,
    then calculates similarity scores using TF-IDF vectorization and cosine similarity
    between the textual representations of books (title + subjects + author names).

    Args:
        target_book (str): The title of the book to find similar ones for.
        limit (int): Number of similar books to return.

    Returns:
        List[Dict]: A list of books (as dictionaries) sorted by similarity score in descending order.
                Each result includes a 'score' field indicating the level of similarity (0.0–1.0).
```

```
"""
def doc_to_text(doc: Dict) -> str:
    parts = []
    if t := doc.get('title'):
        parts.append(t)
    for s in doc.get('subject', []):
        if isinstance(s, str):
            parts.append(s)
    for a in doc.get('author_name', []):
        if isinstance(a, str):
            parts.append(a)
    return " ".join(parts)

offset = 0
all_candidates = []
seen_keys = set()
target_doc = None

while len(all_candidates) < limit:
    batch = find_similar_books(target_book, limit=limit, offset=offset)
    if not batch:
        break

    if offset == 0 and batch:
        target_doc = getattr(find_similar_books, "_target_cache", None)

    for doc in batch:
        key = doc.get("key")
        if key and key not in seen_keys:
            seen_keys.add(key)
            all_candidates.append(doc)
            if len(all_candidates) == limit:
                break

    offset += len(batch)

if not target_doc or not all_candidates:
    return []

texts = [doc_to_text(target_doc)] + [doc_to_text(doc) for doc in all_candidates]
vectorizer = TfidfVectorizer()
```

```
    tfidf = vectorizer.fit_transform(texts)

    sims = cosine_similarity(tfidf[0:1], tfidf[1:]).flatten()

    scored = []
    for doc, sim in zip(all_candidates, sims):
        scored.append({**doc, "score": float(sim)})

    scored.sort(key=lambda x: x["score"], reverse=True)
    return scored[:limit]
```

## filters.py:

```
from typing import List, Dict
from pathlib import Path
from backend.api.open_library_api import get_author_details, logger, get_book_details, find_similar_books, \
    diploma_session, OL_AUTHOR_WORKS_URL, OL_AUTHORS_URL, OL_SEARCH_URL
from backend.models import Book, Author
from fastapi import APIRouter, HTTPException, Query
from starlette.responses import FileResponse
from langdetect import detect, LangDetectException
from backend.recommendation.recommender import (
    recommend_by_genre,
    recommend_similar_authors,
    recommend_similar_books
)
from backend.config import (
    BOOKS_MIN_RATING_DEFAULT,
    BOOKS_MIN_REVIEWS_DEFAULT,
    BOOKS_PAGE_LIMIT_DEFAULT,
    BOOKS_OFFSET_DEFAULT,
    AUTHOR_LIMIT_DEFAULT,
)
from pydantic import BaseModel

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

PROJECT_ROOT = Path(__file__).resolve().parents[2]
```

```
_recs_cache: Dict[str, List[Dict]] = {}
MAX_REC = 200

#-----------------------------------------------------------------------------------------------------------
#                              Routers for genre-based filter
#-----------------------------------------------------------------------------------------------------------

@router.get("/genre_filter")
def get_genre_filter():
    return FileResponse(PROJECT_ROOT / "frontend" / "templates" / "genre_based.html")


@router.get("/api/genre_filter", response_model=List[Book])
async def genre_filter_api(
    genres: List[str] = Query(..., description="List of genres to filter by"),
    min_rating: float = Query(
        BOOKS_MIN_RATING_DEFAULT, description="Minimum average rating"
    ),
    min_reviews: int = Query(
        BOOKS_MIN_REVIEWS_DEFAULT, description="Minimum number of reviews"
    ),
    offset: int = Query(
        BOOKS_OFFSET_DEFAULT, ge=0, description="Pagination offset"
    ),
    limit: int = Query(
        BOOKS_PAGE_LIMIT_DEFAULT, gt=0, le=100, description="Number of books to return"
    ),
):
    return recommend_by_genre(
        genres=genres,
        min_rating=min_rating,
        min_reviews=min_reviews,
        limit=limit,
        offset=offset,
    )


#-----------------------------------------------------------------------------------------------------------
#                              Routers for book details
#-----------------------------------------------------------------------------------------------------------
```

```
@router.get("/book/{work_key}")
def book_detail_page(work_key: str):
    return FileResponse(PROJECT_ROOT / "frontend" / "templates" / "book_details.html")


class BookDetailResponse(BaseModel):
    detail: dict
    recommendations: List[Book]


@router.get("/api/book/{work_key}", response_model=BookDetailResponse)
async def book_detail_api(
        work_key: str,
        limit: int = Query(5, ge=1, le=20),
):
    meta = get_book_details(work_key)
    if not meta:
        raise HTTPException(404, f'No work found for key {work_key}')

    raw = find_similar_books(meta["title"], limit=limit)

    recs = [
        Book(
            key=b["key"],
            title=b["title"],
            authors=b.get("authors") or b.get("author_name", []),
            cover_id=b.get("cover_id") or b.get("cover_i"),
            rating=b.get("rating") or b.get("ratings_average"),
            rating_count=b.get("rating_count") or b.get("ratings_count"),
            genres=b.get("subjects") or b.get("subject", [])
        )
        for b in raw
    ]

    return BookDetailResponse(detail=meta, recommendations=recs)

#-----------------------------------------------------------------------------------------------------------
#                                Routers for author-based filter
#-----------------------------------------------------------------------------------------------------------
```

```
@router.get("/author_filter")
def get_author_filter():
    return FileResponse(PROJECT_ROOT / "frontend" / "templates" / "author_based.html")


@router.get("/api/author", response_model=List[Author])
async def get_author_recommendations(
    author: str = Query(..., min_length=2),
    limit: int = Query(AUTHOR_LIMIT_DEFAULT, ge=1, le=20)
):
    authors = recommend_similar_authors(author, limit=limit)
    return authors[:limit]


@router.get("/api/author/{author_key}/works", response_model=List[Book])
async def get_author_works(
        author_key: str,
        limit: int = Query(5, ge=1, le=20),
        languages: str = Query("en,ru", description="Filter by title languages (comma-separated)")
):
    try:
        allowed_langs = [lang.strip() for lang in languages.split(",")]
        url = OL_AUTHOR_WORKS_URL.format(author_key=author_key)
        params = {
            'limit': 50,
            'fields': 'title,key,cover_i,covers,ratings_average,first_publish_year'
        }
        resp = diploma_session.get(url, params=params, timeout=10)
        resp.raise_for_status()
        data = resp.json()

        books = []
        for work in data.get('entries', []):
            title = work.get('title')
            if not title:
                continue

            try:
                title_lang = detect(title)
            except LangDetectException:
                title_lang = None
```

```
            if title_lang not in allowed_langs:
                continue

            cover_id = work.get('covers', [None])[0] or work.get('cover_i')

            books.append(Book(
                key=work.get('key', '').split('/')[-1],
                title=title,
                cover_id=cover_id,
                rating=work.get('ratings_average')
            ))

            if len(books) >= limit:
                break

        logger.info(f"Returning {len(books)} books filtered by languages {allowed_langs}")
        return books

    except Exception as e:
        logger.error(f"Error fetching works: {str(e)}", exc_info=True)
        raise HTTPException(500, "Failed to fetch author's works")


#-----------------------------------------------------------------------------------------------------------------
#                                Routers for author details
#-----------------------------------------------------------------------------------------------------------------
@router.get("/author/{author_key}")
def author_detail_page(author_key: str):
    return FileResponse(PROJECT_ROOT / "frontend" / "templates" / "author_details.html")


@router.get("/api/author/{author_key}", response_model=Author)
async def get_author_details_api(author_key: str):
    author = get_author_details(author_key)
    if not author:
        raise HTTPException(404, f"No author found for key {author_key}")
    return author


#-----------------------------------------------------------------------------------------------------------------
```

```
#                          Routers for book-based filter
#-------------------------------------------------------------------------------------------------------------

@router.get("/book_filter")
def book_filter():
    return FileResponse(PROJECT_ROOT / "frontend" / "templates" / "book_based.html")


@router.get("/api/book", response_model=List[Book])
async def get_similar_books(
    book: str = Query(..., min_length=2),
    offset: int = Query(0, ge=0),
    limit: int = Query(20, ge=1, le=50),
) -> List[Book]:

    if offset == 0:
        _recs_cache.pop(book, None)

    if book not in _recs_cache:
        _recs_cache[book] = recommend_similar_books(book, limit=MAX_REC)

    full = _recs_cache[book]
    page = full[offset : offset + limit]

    return [
        Book(
            key=rec["key"],
            title=rec["title"],
            authors=rec.get("author_name", []),
            genres=rec.get("subject", []),
            cover_id=rec.get("cover_i"),
            rating=rec.get("ratings_average"),
            rating_count=rec.get("ratings_count"),
        )
        for rec in page
    ]


#-------------------------------------------------------------------------------------------------------------
#                          Router for search autocomplete
#-------------------------------------------------------------------------------------------------------------
```

```
@router.get("/api/book_suggest")
async def book_suggestions(
        query: str = Query(..., min_length=2),
        limit: int = Query(5, ge=1, le=10)
):
    try:
        params = {
            'q': query,
            'limit': limit,
            'fields': 'title,author_name,cover_i'
        }
        resp = diploma_session.get(OL_SEARCH_URL, params=params, timeout=5)
        resp.raise_for_status()
        docs = resp.json().get('docs', [])

        return [{
            'title': doc.get('title'),
            'author_name': doc.get('author_name', []),
            'cover_id': doc.get('cover_i')
        } for doc in docs if doc.get('title')]
    except Exception as e:
        logger.error(f'Error fetching book suggestions: {e}')
        return []


@router.get("/api/author_suggest")
async def author_suggestions(
        query: str = Query(..., min_length=2),
        limit: int = Query(5, ge=1, le=10)
):
    try:
        params = {
            'q': query,
            'limit': limit,
            'fields': 'name,key'
        }
        resp = diploma_session.get(OL_AUTHORS_URL, params=params, timeout=5)
        resp.raise_for_status()
        docs = resp.json().get('docs', [])

        return [{'name': doc['name']} for doc in docs if 'name' in doc]
```

```
    except Exception as e:
        logger.error(f'Error fetching author suggestions: {e}')
        return []
```

## author_based.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Find Similar Authors</title>
  <link rel="stylesheet" href="/static/css/main.css">
  <link rel="stylesheet" href="/static/css/user_input.css">
  <link rel="stylesheet" href="/static/css/recommendation_results.css">
  <link rel="stylesheet" href="/static/css/details_page.css">

</head>
<body>
  <div class="container">
    <header class="header">
      <button class="back-btn" onclick="window.history.back()">
        ← Back
      </button>
      <button class="home-btn" onclick="window.location.href='/'">
        🏠 Home
      </button>
      <h1>Find Similar Authors</h1>
      <p>Enter an author's name to find similar writers</p>
    </header>

    <main class="main-content">
      <div class="author-search-container">
        <div class="search-box">
          <input type="text" id="authorInput" placeholder="Type an author's name..."/>
          <div id="authorSuggestions" class="suggestions-dropdown"></div>
        </div>
        <button id="searchAuthorBtn" class="author-search-btn">Search</button>
      </div>
```

```
    <div id="authorResults" class="authors-results"></div>
   </main>
  </div>

  <script src="/static/js/search_autocomplete.js"></script>
  <script src="/static/js/results.js"></script>
</body>
</html>
```

## author_details.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Author Details</title>
  <link rel="stylesheet" href="/static/css/main.css">
  <link rel="stylesheet" href="/static/css/details_page.css">
  <link rel="stylesheet" href="/static/css/user_input.css">
</head>
<body>
  <div class="container">
    <button class="back-btn" onclick="window.history.back()">
      ← Back
    </button>
    <button class="home-btn" onclick="window.location.href='/'">
      🏠 Home
    </button>
   <main class="detail-main">
    <div class="author-detail-container">
      <div id="author-detail" class="author-detail">
        <!-- Content will be filled by author_details.js -->
      </div>
    </div>
    <div class="author-books-section">
      <h2>Top Books</h2>
      <div id="author-books" class="books-grid"></div>
    </div>
   </main>
  </div>
```

```
  <script src="/static/js/author_details.js"></script>
</body>
</html>
```

## book_based.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Find Similar Books</title>
  <link rel="stylesheet" href="/static/css/main.css">
  <link rel="stylesheet" href="/static/css/user_input.css">
  <link rel="stylesheet" href="/static/css/recommendation_results.css">
</head>
<body>
  <div class="container">
    <header class="header">
      <button class="back-btn" onclick="window.history.back()">
        ← Back
      </button>
      <button class="home-btn" onclick="window.location.href='/'">
        🏠 Home
      </button>
      <h1>Find Similar Books</h1>
      <p>Enter a book title below and we'll recommend similar reads</p>
    </header>

    <main class="main-content">
      <div class="book-search-container">
        <div class="book-search-box">
          <input
            type="text"
            id="bookTitleInput"
            placeholder="Type a book title..."
          />
          <div id="bookSuggestions" class="suggestions-dropdown"></div>
        </div>
        <button id="searchBtn" class="action-btn">Search</button>
      </div>
```

```
      <div class="results-container">
        <div id="results" class="results-grid"></div>
        <button id="loadMoreBtn" class="load-more-btn" style="display: none;">Load More</button>
      </div>
    </main>
  </div>

  <script src="/static/js/search_autocomplete.js"></script>
  <script src="/static/js/results.js"></script>
</body>
</html>
```

## book_details.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Book Details</title>
  <link rel="stylesheet" href="/static/css/main.css">
  <link rel="stylesheet" href="/static/css/details_page.css">
  <link rel="stylesheet" href="/static/css/user_input.css">
</head>
<body>
  <div class="container">
    <button class="back-btn" onclick="window.history.back()">
      ← Back
    </button>
    <button class="home-btn" onclick="window.location.href='/'">
      🏠 Home
    </button>
    <main class="detail-main">
      <div class="book-detail-container">
        <div id="book-detail" class="book-detail">
        </div>
      </div>
      <div class="recommendations-section">
        <h2>You might also like</h2>
        <div id="recommendations" class="recommendations-grid"></div>
```

```
    </div>
   </main>
  </div>
  <script src="/static/js/book_details.js"></script>
</body>
</html>
```

## genre_based.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Genre Filter</title>
  <link rel="stylesheet" href="/static/css/main.css">
  <link rel="stylesheet" href="/static/css/user_input.css">
  <link rel="stylesheet" href="/static/css/recommendation_results.css">
</head>
<body>
  <div class="container">
    <header class="header">
      <button class="back-btn" onclick="window.history.back()">
        ← Back
      </button>
      <button class="home-btn" onclick="window.location.href='/'">
        🏠 Home
      </button>
      <h1>Select Your Favorite Genres</h1>
      <p>Choose the genres you're interested in and we'll find books for you</p>
    </header>

    <main class="main-content">
      <div class="genre-selection">
        <div id="genre_buttons" class="genre-buttons">
          <button type="button" class="genre-btn" data-genre="fantasy">Fantasy</button>
          <button type="button" class="genre-btn" data-genre="mystery">Mystery</button>
          <button type="button" class="genre-btn" data-genre="thriller">Thriller</button>
          <button type="button" class="genre-btn" data-genre="science fiction">Sci-Fi</button>
          <button type="button" class="genre-btn" data-genre="romance">Romance</button>
```

```
        <button type="button" class="genre-btn" data-genre="history">History</button>
        <button type="button" class="genre-btn" data-genre="biography">Biography</button>
        <button type="button" class="genre-btn" data-genre="nonfiction">Non-Fiction</button>
      </div>

      <div class="action-buttons">
        <button id="continueBtn" disabled>Find Books</button>
      </div>
    </div>
    <div class="results-container">
      <div id="results" class="results-grid"></div>
      <button id="loadMoreBtn" class="load-more-btn" style="display: none;">Load More</button>
    </div>
  </main>
 </div>
 <script src="/static/js/results.js"></script>
</body>
</html>
```

index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Book Discovery Service</title>
    <link rel="stylesheet" href="/static/css/main.css">
    <link rel="stylesheet" href="/static/css/user_input.css">
    <link rel="stylesheet" href="/static/css/recommendation_results.css">
    <link rel="stylesheet" href="/static/css/details_page.css">
</head>
<body>
  <div class="container">
    <header class="header">
      <h1>Book Discovery Service</h1>
      <p class="tagline">Find your next favorite book with our smart recommendations</p>
    </header>

    <main class="main-content">
      <section class="service-description">
```

```
        <h2>Discover Books Tailored to Your Taste</h2>
        <p>Our service helps you find books you'll love through intelligent filtering and recommendation systems. Whether you know exactly what you want or just want to explore, we've got you covered.</p>

        <div class="features">
          <div class="feature-card">
            <h3>Genre Explorer</h3>
            <p>Browse books by genre and discover new favorites in categories you love.</p>
          </div>
          <div class="feature-card">
            <h3>Author Match</h3>
            <p>Find books similar to works by your favorite authors.</p>
          </div>
          <div class="feature-card">
            <h3>Book Companion</h3>
            <p>Get recommendations based on books you've enjoyed.</p>
          </div>
        </div>
      </section>

      <section class="discovery-options">
        <h2>Start Discovering</h2>
        <div class="option-buttons">
          <a href="/genre_filter" class="discovery-btn genre">
            <span class="icon">📚</span>
            <span>Browse by Genre</span>
          </a>
          <a href="/author_filter" class="discovery-btn author">
            <span class="icon">✍️</span>
            <span>Find Similar Authors</span>
          </a>
          <a href="/book_filter" class="discovery-btn book">
            <span class="icon">🔍</span>
            <span>Find Similar Books</span>
          </a>
        </div>
      </section>
    </main>

    <footer class="footer">
      <p>© 2023 Book Discovery Service. All rights reserved.</p>
```

```
      </footer>
    </div>
</body>
</html>
```

## author_details.js:

```
document.addEventListener("DOMContentLoaded", async () => {
 /**
 * Load and display author details and their works when the author page is opened.
 *
 * This script runs after the DOM is fully loaded. It performs the following steps:
 * 1. Extracts the author's key from the URL path.
 * 2. Displays loading indicators in the author and books sections.
 * 3. Fetches the author's metadata from `/api/author/{authorKey}` and renders:
 *    - Name, photo, birth/death dates, works count, rating, and biography.
 * 4. Fetches the author's books from `/api/author/{authorKey}/works?languages=en,ru` and renders:
 *    - Book cover, title, and rating (if available).
 * 5. Handles missing data (e.g., no photo or books) with placeholders and fallback messages.
 * 6. Catches and displays any errors that occur during the fetch process.
 */

 const authorEl = document.getElementById("author-detail");
 const booksEl = document.getElementById("author-books");
 const parts = window.location.pathname.split("/");
 const authorKey = parts[parts.length - 1];

 // Loading states
 authorEl.innerHTML = `
   <div class="loading">
     <div class="loading-spinner"></div>
     <div class="loading-message">Loading author details...</div>
   </div>
 `;
 booksEl.innerHTML = `
   <div class="loading">
     <div class="loading-spinner"></div>
     <div class="loading-message">Loading author's books...</div>
   </div>
 `;
```

```
try {
  // Fetch author details
  const authorResp = await fetch(`/api/author/${authorKey}`);
  if (!authorResp.ok) throw new Error(authorResp.statusText);
  const author = await authorResp.json();

  // Build author detail panel
  let html = `
    <div class="author-header">
      <div class="author-photo-container">
        ${author.photo_id
          ? `<img
              src="https://covers.openlibrary.org/a/olid/${author.key}-L.jpg"
              alt="${author.name}"
              class="author-photo-large">`
          : `<img
              src="/static/images/no-photo-author.jpg"
              alt="Default Author"
              class="author-photo-large placeholder">`
        }
      </div>
      <div class="author-info">
        <h1 class="author-name">${author.name}</h1>

        <div class="author-meta">
          ${author.birth_date ? `<p><strong>Born:</strong> ${author.birth_date}</p>` : ''}
          ${author.death_date ? `<p><strong>Died:</strong> ${author.death_date}</p>` : ''}
          ${author.works_count ? `<p><strong>Works:</strong> ${author.works_count}</p>` : ''}
          ${author.rating ? `<p><strong>Average Rating:</strong> ${author.rating.toFixed(1)}</p>` : ''}
        </div>
      </div>
    </div>

    ${author.bio ? `
      <div class="author-bio">
        <h3>Biography</h3>
        <p>${author.bio}</p>
      </div>
    ` : ''}
  `;
  authorEl.innerHTML = html;
```

```
  // Fetch author's books
  const booksResp = await fetch(`/api/author/${authorKey}/works?languages=en,ru`);
  if (!booksResp.ok) throw new Error(booksResp.statusText);
  const books = await booksResp.json();

  // Build books list
  if (books.length) {
   let booksHtml = books.map(b => `
    <div class="book-card">
     <a href="/book/${b.key.replace("/works/","")}">
      ${b.cover_id
       ? `<img
           src="https://covers.openlibrary.org/b/id/${b.cover_id}-M.jpg"
           alt="${b.title}"
           class="book-cover">`
       : `<img
           src="/static/images/no-cover-photo.jpg"
           alt="Default cover"
           class="book-cover placeholder">`
      }
      <div class="book-info">
       <h4>${b.title}</h4>
       ${b.rating ? `<p class="book-rating">★ ${b.rating.toFixed(1)}</p>` : ''}
      </div>
     </a>
    </div>
   `).join('');
   booksEl.innerHTML = booksHtml;
  } else {
   booksEl.innerHTML = '<p class="empty-message">No books found for this author</p>';
  }

 } catch (err) {
  authorEl.innerHTML = `<p class="error-message">Failed to load: ${err.message}</p>`;
  booksEl.innerHTML = "";
 }
});
```

## book_details.js:

```
document.addEventListener("DOMContentLoaded", async () => {
  /**
 * Load and display book details and recommended books on the book detail page.
 *
 * This script executes when the DOM is fully loaded and performs the following steps:
 * 1. Extracts the book's work key from the current URL path.
 * 2. Shows loading placeholders for both the book detail section and recommendations.
 * 3. Sends a request to `/api/book/{workKey}?limit=5` to retrieve:
 *    - Detailed metadata for the book (title, authors, rating, subjects, description, cover).
 *    - A list of recommended similar books.
 * 4. Renders the book information and its metadata in the `#book-detail` element.
 * 5. Renders up to 5 book recommendations as clickable cards with title, author(s), and cover.
 * 6. Displays appropriate fallback messages when no data is available or if an error occurs.
 */

 const detailEl = document.getElementById("book-detail");
 const recEl = document.getElementById("recommendations");
 const parts = window.location.pathname.split("/");
 const workKey = parts[parts.length - 1];

 detailEl.innerHTML = "<div class='loading'>Loading book details...</div>";
 recEl.innerHTML = "<div class='loading'>Loading recommendations...</div>";

 try {
  const resp = await fetch(`/api/book/${workKey}?limit=5`);
  if (!resp.ok) throw new Error(resp.statusText);
  const { detail, recommendations } = await resp.json();

  // Build detail panel
  let html = `
   <h1 class="book-title">${detail.title}</h1>
   <div class="book-meta-container">
    ${detail.authors.length ? `
     <div class="book-meta">
      <strong>Author:</strong> ${detail.authors.join(", ")}
     </div>
    ` : ''}
    ${detail.rating ? `
     <div class="book-meta">
      <strong>Rating:</strong> ${detail.rating.toFixed(1)} (${detail.rating_count || 0} reviews)
     </div>
```

```
    ` : ''}
    ${detail.subjects?.length ? `
      <div class="book-meta">
        <strong>Genres:</strong> ${detail.subjects.slice(0, 5).join(", ")}
      </div>
    ` : ''}
  </div>
  <div class="book-content">
    ${detail.covers?.length ? `
      <div class="book-cover-container">
        <img src="https://covers.openlibrary.org/b/id/${detail.covers[0]}-L.jpg"
            alt="Cover for ${detail.title}" class="book-cover">
      </div>
    ` : ''}
    <div class="book-description">
      <h3>Description</h3>
      <p>${detail.description || 'No description available.'}</p>
    </div>
  </div>
`;
detailEl.innerHTML = html;

// Build recommendation list
if (recommendations.length) {
  let recHtml = recommendations.map(b => `
    <div class="recommendation-card">
      <a href="/book/${b.key.replace("/works/","")}">
        ${b.cover_id ?
          `<img src="https://covers.openlibrary.org/b/id/${b.cover_id}-M.jpg"
              alt="${b.title}" class="rec-cover">` : ''}
        <h4>${b.title}</h4>
        <p>${b.authors?.join(", ") || "Unknown author"}</p>
      </a>
    </div>
  `).join('');
  recEl.innerHTML = recHtml;
} else {
  recEl.innerHTML = '<p>No recommendations found</p>';
}

} catch (err) {
```

```
    detailEl.innerHTML = `<p class="error">Failed to load: ${err.message}</p>`;
    recEl.innerHTML = "";
  }
});
```

## results.js:

```
/* ------------------------------------------------------------------------------------------------------------
---------------------------------------- Genre-based ----------------------------------------------------------
------------------------------------------------------------------------------------------------------------ */

document.addEventListener("DOMContentLoaded", () => {
  /**
  * Enable genre-based book discovery with dynamic filtering and pagination.

  * This script attaches interactive behavior to the genre selection interface
  * and fetches books from the server based on selected genres.

  * Functionality includes:
  * 1. Tracks selected genres using button toggles.
  * 2. Enables or disables the "Find Books" button depending on whether any genres are selected.
  * 3. On "Find Books" click:
  *    - Resets current state.
  *    - Sends an API request with selected genres and fetches the first page of books.
  *    - Displays results in a card-based layout with title, authors, cover, and rating.
  * 4. On "Load More" click:
  *    - Fetches the next set of results using an updated offset.
  *    - Appends the new results without clearing previous ones.
  * 5. Dynamically hides or shows the "Load More" button depending on whether more results are available.
  * 6. Handles empty results and errors with user-friendly messages.
  */

  const genreButtons = document.querySelectorAll(".genre-btn");
  const continueBtn = document.getElementById("continueBtn");
  const loadMoreBtn = document.getElementById("loadMoreBtn");
  const out = document.getElementById("results");

  let selectedGenres = new Set();
  let offset = 0;
  const limit = 20;
```

```
// Create a standardized book card DOM element
function createBookCard(book) {
  const card = document.createElement("div");
  card.className = "book-card";

  const link = document.createElement("a");
  link.href = `/book/${book.key.replace("/works/", "")}`;
  link.className = "book-link";

  if (book.cover_id) {
    const img = document.createElement("img");
    img.src = `https://covers.openlibrary.org/b/id/${book.cover_id}-M.jpg`;
    img.alt = `Cover for ${book.title}`;
    img.className = "book-cover";
    img.loading = "lazy";

    const wrapper = document.createElement("div");
    wrapper.className = "cover-wrapper";
    wrapper.appendChild(img);
    link.appendChild(wrapper);
  } else {
    const placeholder = document.createElement("div");
    placeholder.className = "book-cover placeholder";
    link.appendChild(placeholder);
  }

  const info = document.createElement("div");
  info.className = "book-info";

  const h3 = document.createElement("h3");
  h3.textContent = book.title || "No title available";
  info.appendChild(h3);

  const pAuthor = document.createElement("p");
  pAuthor.textContent = (book.authors || []).join(", ") || "Unknown author";
  pAuthor.className = "book-authors";
  info.appendChild(pAuthor);

  if (book.rating) {
    const pRating = document.createElement("p");
```

```
    pRating.textContent = `★ ${book.rating.toFixed(1)}`;
    pRating.className = "book-rating";
    info.appendChild(pRating);
  }

  link.appendChild(info);
  card.appendChild(link);

  return card;
}

// Handle genre button selection/deselection
genreButtons.forEach(btn => {
  btn.addEventListener("click", () => {
    const genre = btn.dataset.genre;
    if (selectedGenres.has(genre)) {
      selectedGenres.delete(genre);
      btn.classList.remove("selected");
    } else {
      selectedGenres.add(genre);
      btn.classList.add("selected");
    }
    continueBtn.disabled = selectedGenres.size === 0;
  });
});

// Handle "Find Books" click — reset state and load first page
continueBtn.addEventListener("click", () => {
  offset = 0;
  out.innerHTML = "";
  loadMoreBtn.style.display = "none";
  fetchBooks();
});

// Handle "Load More" click — fetch next page
loadMoreBtn.addEventListener("click", () => {
  loadMoreBtn.disabled = true;
  fetchBooks().then(() => {
    loadMoreBtn.disabled = false;
  });
});
```

```
// Fetch books from the API using current genre selections and offset
async function fetchBooks() {
  const genresArray = Array.from(selectedGenres);
  const params = new URLSearchParams();
  genresArray.forEach(g => params.append("genres", g));
  params.append("offset", offset);
  params.append("limit", limit);

  // Show loading animation for initial search
  if (offset === 0) {
    out.innerHTML = `
      <div class="loading-container">
        <div class="loading">
          <div class="loading-spinner"></div>
          <div class="loading-message">Searching for books...</div>
        </div>
      </div>`;
  }

  try {
    const resp = await fetch(`/api/genre_filter?${params.toString()}`);
    if (!resp.ok) throw new Error(`Error ${resp.status}`);
    const books = await resp.json();

    if (offset === 0 && books.length === 0) {
      out.innerHTML = '<div class="empty-message">No books found matching your selected genres.</div>';
      return;
    }

    if (offset === 0) {
      out.innerHTML = "";
    }

    books.forEach(book => {
      out.appendChild(createBookCard(book));
    });

    offset += books.length;

    // Show or hide Load More button depending on result size
```

```
    loadMoreBtn.style.display = books.length < limit ? "none" : "block";

  } catch (err) {
    if (offset === 0) {
      out.innerHTML = `<div class="error-message">Error: ${err.message}</div>`;
    } else {
      loadMoreBtn.style.display = "none";
      console.error(err);
    }
  }
 }

 // Hide Load More button on initial page load
 loadMoreBtn.style.display = "none";
});
```

```
/* ---------------------------------------------------------------------------------------------------------
---------------------------------------- Author-based ----------------------------------------------------
--------------------------------------------------------------------------------------------------------- */

document.addEventListener("DOMContentLoaded", () => {
 /**
 * Enable interactive author search and display of similar authors based on user input.

 * This script is triggered when the DOM is fully loaded and provides the following functionality:
 * 1. Validates the author's name entered by the user in the input field.
 * 2. On "Search" button click:
 *    - Sends a request to `/api/author?author={authorName}` to fetch similar authors.
 *    - Displays a loading animation while waiting for the response.
 *    - If results are found:
 *        • Renders a grid of author cards with photo, name, known work, and average rating.
 *    - If no results are found:
 *        • Displays a friendly message indicating no similar authors were found.
 * 3. Handles invalid input (less than 2 characters) and server-side or network errors with clear feedback.
 */
  const searchAuthorBtn = document.getElementById("searchAuthorBtn");
  const authorInput = document.getElementById("authorInput");
  const authorResults = document.getElementById("authorResults");
```

```
if (searchAuthorBtn && authorInput && authorResults) {
    searchAuthorBtn.addEventListener("click", async () => {
        const authorName = authorInput.value.trim();

        // Input validation
        if (authorName.length < 2) {
            authorResults.innerHTML = '<div class="error-message">Please enter at least 2 characters</div>';
            return;
        }

            authorResults.innerHTML = `
                <div class="loading">
                    <div class="loading-spinner"></div>
                    <div class="loading-message">Searching for authors...</div>
                </div>
            `;


        try {
            const response = await fetch(`/api/author?author=${encodeURIComponent(authorName)}`);

            if (!response.ok) {
                const error = await response.json();
                throw new Error(error.detail || "Server error");
            }

            const authors = await response.json();

            if (!authors.length) {
                authorResults.innerHTML = `
                    <div class="empty-message">
                        No similar authors found for "${authorName}"
                    </div>
                `;
                return;
            }

            // In the author search results section
            authorResults.innerHTML = `
            <h3>Authors similar to ${authorName}:</h3>
```

```
            <div class="authors-grid">
            ${authors.map(author => `
              <div class="author-card">
                <a href="/author/${author.key}">
                  ${author.photo_id
                    ? `<img src="https://covers.openlibrary.org/a/olid/${author.key}-M.jpg"
                      alt="${author.name}" class="author-photo">`
                    : `<img src="/static/images/no-photo-author.jpg"
                      alt="Default author image" class="author-photo placeholder">`
                  }
                <h4>${author.name}</h4>
                ${author.top_work ? `<p class="author-work">Known for: ${author.top_work}</p>` : ''}
                ${author.rating ? `<p class="author-rating">Avg rating: ${author.rating.toFixed(1)}</p>` : ''}
                </a>
              </div>
            `).join('')}
            </div>
            `;
          } catch (err) {
            authorResults.innerHTML = `
              <div class="error-message">
                Error: ${err.message || "Unknown error occurred"}
              </div>
            `;
          }
        });
    }
});


/* ---------------------------------------------------------------------------------------------------------
---------------------------------------- Book-based ----------------------------------------------------------
--------------------------------------------------------------------------------------------------------- */

document.addEventListener("DOMContentLoaded", () => {
  /**
  * Enable dynamic book search and paginated loading of similar book recommendations.

  * This script allows users to:
  * - Search for books based on a title input.
  * - Fetch the initial result set and render it as a grid of book cards.
```

```
* - Load additional pages using the "Load More" button.
* - Handle errors, loading states, and display fallback messages for empty results.
*/
 const bookInput       = document.getElementById("bookTitleInput");
 const searchBtn       = document.getElementById("searchBtn");
 const loadMoreBtn     = document.getElementById("loadMoreBtn");
 const resultsContainer = document.getElementById("results");

 if (!bookInput || !searchBtn) return;

 let offset      = 0;
 const pageSize  = 20;
 let currentQuery = "";

 loadMoreBtn.style.display  = "none";
 loadMoreBtn.disabled       = true;

 async function fetchBooks(reset = false) {
   const title = bookInput.value.trim();
   if (!title) return;

   if (reset) {
     offset = 0;
     currentQuery = title;
     resultsContainer.innerHTML = "";
     loadMoreBtn.style.display = "none";
     loadMoreBtn.disabled      = true;
   } else {
     loadMoreBtn.disabled      = true;
     loadMoreBtn.classList.add("loading");
   }

   if (reset) {
     resultsContainer.innerHTML = `
       <div class="loading-container">
         <div class="loading">
           <div class="loading-spinner"></div>
           <div class="loading-message">Searching for books…</div>
         </div>
       </div>`;
   }
```

```
try {
  const url = `/api/book?book=${encodeURIComponent(currentQuery)}&offset=${offset}&limit=${pageSize}`;
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error(`Server returned ${response.status}`);
  }
  const books = await response.json();

  if (reset && books.length === 0) {
    resultsContainer.innerHTML = `
      <div class="empty-message">
        No similar books found for “${currentQuery}”
      </div>`;
    return;
  }

  if (reset) {
    resultsContainer.innerHTML = "";
  }

  books.forEach(book => {
    const card = document.createElement("div");
    card.className = "book-card";

    const link = document.createElement("a");
    link.href      = `/book/${book.key.replace("/works/","")}`;
    link.className = "book-link";

    if (book.cover_id) {
      const wrapper = document.createElement("div");
      wrapper.className = "cover-wrapper";

      const img = document.createElement("img");
      img.src       = `https://covers.openlibrary.org/b/id/${book.cover_id}-M.jpg`;
      img.alt       = `Cover for ${book.title}`;
      img.className = "book-cover";
      img.loading   = "lazy";

      wrapper.appendChild(img);
```

```
    link.appendChild(wrapper);
  } else {
    const placeholder = document.createElement("div");
    placeholder.className = "book-cover placeholder";
    link.appendChild(placeholder);
  }

  const info = document.createElement("div");
  info.className = "book-info";

  const h3 = document.createElement("h3");
  h3.textContent = book.title || "No title available";
  info.appendChild(h3);

  const pAuthor = document.createElement("p");
  pAuthor.className = "book-authors";
  pAuthor.textContent = (book.authors?.join(", ") || "Unknown author");
  info.appendChild(pAuthor);

  if (book.rating != null) {
    const pRating = document.createElement("p");
    pRating.className = "book-rating";
    pRating.textContent = `★ ${book.rating.toFixed(1)}`;
    info.appendChild(pRating);
  }

  link.appendChild(info);
  card.appendChild(link);
  resultsContainer.appendChild(card);
});

if (books.length === pageSize) {
  loadMoreBtn.style.display = "block";
} else {
  loadMoreBtn.style.display = "none";
  const endMsg = document.createElement("div");
  endMsg.className = "end-message";
  endMsg.textContent = "No more books found.";
  resultsContainer.appendChild(endMsg);
}
```

```
    offset += books.length;

   } catch (err) {
    resultsContainer.innerHTML = "";
    const errDiv = document.createElement("div");
    errDiv.className = "error-message";
    errDiv.textContent = `Error: ${err.message}`;
    resultsContainer.appendChild(errDiv);
    loadMoreBtn.style.display = "none";

   } finally {
    loadMoreBtn.disabled = false;
    loadMoreBtn.classList.remove("loading");
   }
  }

  searchBtn.addEventListener("click", () => fetchBooks(true));
  bookInput.addEventListener("keydown", e => {
   if (e.key === "Enter") fetchBooks(true);
  });
  loadMoreBtn.addEventListener("click", () => fetchBooks(false));
});
```

# search_autocomplete.js:

```
/**
 * Enable dynamic autocomplete suggestions for author and book inputs.

 * This script adds live suggestion functionality to input fields for authors and books:
 *
 * Features:
 * 1. Debounced API calls to avoid spamming the server while the user is typing.
 * 2. When the user types at least 2 characters in the input:
 *    - `/api/author_suggest?query=...` returns matching authors.
 *    - `/api/book_suggest?query=...` returns matching books.
 * 3. Suggestions are displayed below the respective input:
 *    - For authors: simple text list with clickable names.
 *    - For books: includes cover thumbnail, title, and author name.
 * 4. Clicking on a suggestion autofill the input field and triggers a corresponding search button.
 * 5. Suggestion lists are hidden:
```

```
*    - When the input is cleared or too short.
*    - When clicking outside the input/suggestions area.
*/
 const authorInput = document.getElementById("authorInput");
 const authorSuggestions = document.getElementById("authorSuggestions");
 const bookInput = document.getElementById("bookTitleInput");
 const bookSuggestions = document.getElementById("bookSuggestions");

// Debounce function to limit API calls
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => func.apply(this, args), delay);
  };
}
/* ----------------------------------------------------------------------------------------------------------
// ---------------------------------------------- Author suggestions ----------------------------------------
----------------------------------------------------------------------------------------------------------*/

if (authorInput && authorSuggestions) {
  // Fetch author suggestions
  async function fetchAuthorSuggestions(query) {
    if (query.length < 2) {
      authorSuggestions.style.display = 'none';
      return;
    }

    try {
      const response = await fetch(`/api/author_suggest?query=${encodeURIComponent(query)}`);
      if (!response.ok) {
        console.error('Author suggestions API error:', response.status);
        return;
      }
      const authors = await response.json();
      console.log('Author suggestions response:', authors); // Debug log
      displayAuthorSuggestions(authors);
    } catch (err) {
      console.error('Error fetching author suggestions:', err);
    }
  }
```

```
// Display author suggestions
function displayAuthorSuggestions(authors) {
  if (!authors || authors.length === 0) {
    authorSuggestions.style.display = 'none';
    return;
  }

  authorSuggestions.innerHTML = authors.map(author => `
    <div class="suggestion-item" data-name="${author.name}">
      ${author.name}
    </div>
  `).join('');

  authorSuggestions.style.display = 'block';

  // Add click handlers to suggestions
  document.querySelectorAll('#authorSuggestions .suggestion-item').forEach(item => {
    item.addEventListener('click', () => {
      authorInput.value = item.dataset.name;
      authorSuggestions.style.display = 'none';
      const searchBtn = document.getElementById("searchAuthorBtn");
      if (searchBtn) searchBtn.click();
    });
  });
}

// Set up event listeners for author input
authorInput.addEventListener('input', debounce(() => {
  fetchAuthorSuggestions(authorInput.value.trim());
}, 300));

// Show suggestions when focusing input
authorInput.addEventListener('focus', () => {
  if (authorInput.value.trim().length >= 2) {
    fetchAuthorSuggestions(authorInput.value.trim());
  }
});
}

/* ----------------------------------------------------------------------------------------------------
```

```
// ---------------------------------------- Book suggestions ----------------------------------------
---------------------------------------------------------------------------------------------------------------*/
if (bookInput && bookSuggestions) {
  // Fetch book suggestions
  async function fetchBookSuggestions(query) {
    if (query.length < 2) {
      bookSuggestions.style.display = 'none';
      return;
    }

    try {
      const response = await fetch(`/api/book_suggest?query=${encodeURIComponent(query)}`);
      if (!response.ok) {
        console.error('Book suggestions API error:', response.status);
        return;
      }
      const books = await response.json();
      console.log('Book suggestions response:', books); // Debug log
      displayBookSuggestions(books);
    } catch (err) {
      console.error('Error fetching book suggestions:', err);
    }
  }

  // Display book suggestions
  function displayBookSuggestions(books) {
    if (!books || books.length === 0) {
      bookSuggestions.style.display = 'none';
      return;
    }

    bookSuggestions.innerHTML = books.map(book => `
      <div class="suggestion-item" data-title="${book.title}">
        ${book.cover_id ?
          `<img src="https://covers.openlibrary.org/b/id/${book.cover_id}-S.jpg"
                alt="${book.title}" class="suggestion-cover">` :
          `<div class="suggestion-cover placeholder"></div>`}
        <div class="suggestion-text">
          <div class="suggestion-title">${book.title}</div>
          ${book.author_name ? `<div class="suggestion-author">${book.author_name[0]}</div>` : ''}
        </div>
```

```
      </div>
    `).join('');

    bookSuggestions.style.display = 'block';

    // Add click handlers to suggestions
    document.querySelectorAll('#bookSuggestions .suggestion-item').forEach(item => {
      item.addEventListener('click', () => {
        bookInput.value = item.dataset.title;
        bookSuggestions.style.display = 'none';
        const searchBtn = document.getElementById("searchBtn");
        if (searchBtn) searchBtn.click();
      });
    });
  }

  // Set up event listeners for book input
  bookInput.addEventListener('input', debounce(() => {
    fetchBookSuggestions(bookInput.value.trim());
  }, 300));

  // Show suggestions when focusing input
  bookInput.addEventListener('focus', () => {
    if (bookInput.value.trim().length >= 2) {
      fetchBookSuggestions(bookInput.value.trim());
    }
  });
}

/* ---------------------------------------------------------------------------------------------------------
// ----------------------------------------- Outside click handler ------------------------------------------
--------------------------------------------------------------------------------------------------------- */
document.addEventListener('click', (e) => {
  // Hide author suggestions if clicking outside
  if (authorSuggestions && !authorInput.contains(e.target) && !authorSuggestions.contains(e.target)) {
    authorSuggestions.style.display = 'none';
  }

  // Hide book suggestions if clicking outside
  if (bookSuggestions && !bookInput.contains(e.target) && !bookSuggestions.contains(e.target)) {
```

```
    bookSuggestions.style.display = 'none';
  }


}
```