# Library Manager Application: Backend–Frontend Implementation

**Name: Farhan Akthar Ponnari**

**Matriculation No: 100004698**

**SRH UNIVERSITY**

**Course Name: Advanced Programming**

**Instructor: Esteban Pozo**

**Due Date: December 12,2025**

## Abstract

This project implements a small “online library” application consisting of a Flask-based backend and a Tkinter (ttk) desktop GUI frontend. The backend stores media metadata (books, films, magazines) in a JSON file and exposes REST endpoints for listing, filtering, searching, creating, updating, and deleting media items. The frontend consumes those endpoints and provides a user-friendly GUI for interacting with the library. This document describes requirements and architecture, explains design and implementation decisions, presents tests, discusses limitations and potential improvements, and includes an annex covering the use of AI tools and prompts. The implementation follows the assignment specification and uses persistent JSON storage to meet the course requirements.

## Contents

## Introduction

This project focuses on the development of a library management application that integrates a Flask-based backend with a Tkinter graphical user interface (GUI) frontend. The objective of the assignment is to demonstrate an understanding of software architecture, client–server communication, data validation, persistent storage, and graphical interface design. Throughout the development process, the project follows best programming practices, including modular code structure, separation of concerns, and the use of RESTful endpoints to connect the frontend with the backend.

This report provides a detailed explanation of the design decisions, implementation steps, testing procedures, and challenges encountered. It also highlights how AI-generated code was used responsibly and adjusted where needed to meet the assignment requirements and ensure correct functionality.

The graphical interface allows users to filter items, search by name, and manage media entries and edit. An example of the interface is shown below.

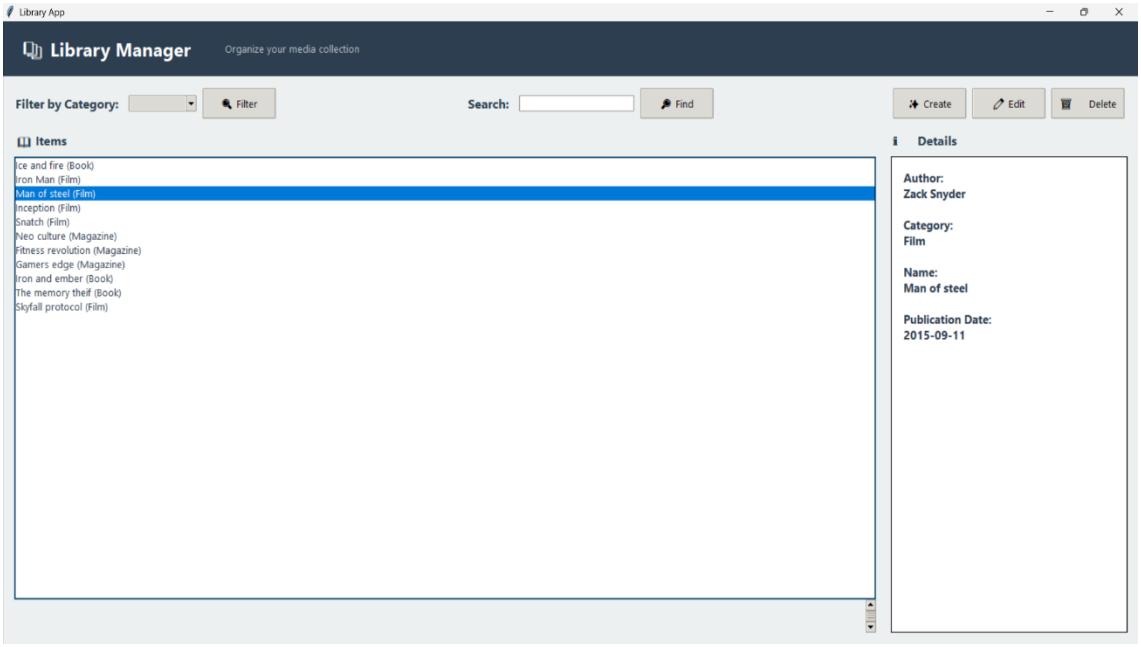


*Figure 1: Graphical user interface of the library application.*

The backend stores all items in a local JSON file, which ensures persistent storage without requiring a database system. A sample structure is shown below.


```
{
  "a32d5fe1-8e53-4794-9f79-c2e2938ea1e5": {
    "id": "a32d5fe1-8e53-4794-9f79-c2e2938ea1e5",
    "name": "Ice and fire",
    "publication_date": "1996-11-25",
    "author": "RR Martin",
    "category": "Book"
  },
  "1c582ed2-b13d-41be-b843-2ef64e179336": {
    "id": "1c582ed2-b13d-41be-b843-2ef64e179336",
    "name": "Iron Man",
    "publication_date": "2010-02-25",
    "author": "James Gunn",
    "category": "Film"
```


*Figure 2: Example of the JSON data structure used for media storage.*

## Task 1: Concept, Requirements, and Architecture

### Functional Requirements

The application must store three categories of media items: books, films, and magazines. Each item contains a name, publication date, author, category, and a unique identifier. The backend must provide endpoints that allow clients to list all items, filter by category, search by exact name, retrieve a specific record, create new entries, update existing entries, and delete records. The frontend must allow users to perform all these operations through a graphical interface.

### Non-Functional Requirements

The system must include persistent storage using a JSON file. It must validate user input, handle errors gracefully, and follow a modular design. The user interface should be simple and intuitive. The backend and frontend must communicate using structured JSON over HTTP.

### Architecture Overview

The application uses a client–server model. The backend, built with Flask, processes requests and interacts with the JSON storage module. The frontend, built with Tkinter, sends HTTP requests to the backend and displays responses. The storage module (`storage.py`) is responsible for loading and saving the JSON file and performing validation on all fields.

Data exchanged between the frontend and backend is formatted as JSON. For example, a media item contains fields such as `id`, `name`, `publication_date`, `author`, and `category`.

### GUI Concepts

The GUI consists of a header section, filter and search controls, a list of media items on the left, and a details panel on the right. Users can create, edit, or delete items using dialog windows. Selecting an item in the list displays its metadata. Errors such as missing fields or invalid date formats are shown through message boxes.

### Anticipated Errors

Possible errors include invalid input formats, missing fields, incorrect JSON structure, server connection failures, and unexpected exceptions during file operations. These are handled through backend validation, try-except blocks, and user-friendly GUI warnings.

## Task 2: Implementation

### Backend Implementation

Backend tests verify that the API endpoints work correctly. These include creating an item, retrieving it by ID, updating it, deleting it, listing all items, filtering by category, and searching by exact name. The tests use the `requests` library to interact with the running backend, ensuring that the endpoints return the intended responses and status codes.

### Frontend Implementation

The frontend is implemented in the `gui.py` file using Tkinter and ttk widgets. The main window includes category filters, a search bar, a listbox for displaying items, and a details panel. Dialog windows are used for creating and editing items, each with input fields and validation. The frontend sends HTTP requests to the backend using the

`requests` library and handles connection errors and invalid input through message boxes.

The interface updates dynamically whenever items are created, edited, or deleted. Validation ensures that all fields are filled and that the publication date follows the correct format before sending data to the backend.

**Design Decisions**

JSON was chosen for storage due to its simplicity, readability, and suitability for small applications. Tkinter was selected because it is included in Python's standard library and supports rapid GUI development. The separation between frontend, backend, and data handling ensures modularity and allows for easier debugging and future modifications.

## Task 3: Tests

**Backend Tests**

Backend tests verify that the API endpoints work correctly. These include creating an item, retrieving it by ID, updating it, deleting it, listing all items, filtering by category, and searching by exact name. The tests use the `requests` library to interact with the running backend, ensuring that the endpoints return the intended responses and status codes.

**Frontend Tests**

A smoke test was performed to ensure that the frontend can successfully communicate with the backend. The smoke test simulates operations such as loading the item list, creating a new entry, editing it, and deleting it. Although full automated GUI

testing was not implemented, the smoke test confirms that communication between the frontend and backend works correctly.

## Task 4: Discussion

### Strengths

The application fulfills all assignment requirements and provides a complete, functional solution. The clear separation between frontend and backend improves maintainability. Input validation on both sides minimizes errors. The GUI is user-friendly and includes helpful feedback messages. JSON storage is simple and effective for a small project.

### Weaknesses and Limitations

JSON is not ideal for large datasets or multi-user scenarios. There is no authentication, meaning any client can modify data. GUI automation tests are limited. Error logging in the backend could be more detailed. The GUI relies on blocking network calls, which may freeze the interface briefly during slow responses.

### Future Improvements

Replacing the JSON file with a database such as SQLite would improve scalability. Adding authentication would increase security. Implementing asynchronous requests could improve responsiveness. Automated GUI tests could help detect interface issues. Additional features, such as sorting, pagination, or exporting data, could enhance usability.

## References

SRH University. (2025). *Advanced programming project description*.

SRH University. (2025). *Project template for academic report*.

Rojas, A. (2020). *Zombies: How to survive the apocalypse*. National American University.

## Annex- AI Usage,Prompts and Corrections

This project made use of AI-assisted development tools to support code generation, error resolution, and explanation of programming concepts. The AI tools used were ChatGPT and GitHub Copilot. The following section documents representative prompts, outputs, and the corrections made during implementation, in accordance with the assignment requirements.

**1. Use of ChatGPT**

**Prompt 1 (to ChatGPT):**

"Help me create a Flask backend that stores books, films, and magazines in JSON format. I need endpoints for listing all items, filtering by category, searching by exact name, creating an item, updating an item, and deleting an item."

**AI Output 1 (excerpt):**

ChatGPT returned a basic Flask structure with @app.route definitions for GET, POST, and DELETE operations. It also suggested using a global dictionary to store items temporarily and returning JSON responses using jsonify().

**Corrections Made:**

- Modified the AI suggestion to use a **separate storage module (storage.py)** instead of storing data in memory.
- Added **UUID generation** for item IDs.
- Added **input validation** for empty fields and incorrect date formats.
- Implemented **persistent saving** to a JSON file rather than keeping data in memory as the AI initially proposed.

**Prompt 2 (to ChatGPT):**

"I need a Tkinter GUI that connects to my Flask backend and allows filtering, searching, creating, editing, and deleting media items. Include error handling and make it look modern."

**AI Output 2 (excerpt):**

ChatGPT generated a Tkinter window with basic buttons and listboxes, including sample functions calling requests.get() and requests.post().

**Corrections Made:**

- Improved the interface styling using ttk themes and a custom color scheme.
- Added **modal dialog windows** for Create and Edit actions (AI only suggested basic input dialogs).
- Added **regular expression validation** for the publication date.
- Implemented **scrollbars**, better layout, and text formatting in the details panel.
- Added explicit **error messages** when the backend is unreachable.

**Prompt 3 (to ChatGPT):**

"Give me example test cases for my Flask backend, including create, retrieve, update, delete, filter, and search functionality."

**AI Output 3 (excerpt):**

AI suggested basic unit tests using Python's unittest module and sending HTTP requests to verify the responses.

**Corrections Made:**

- Adjusted endpoints to match my exact API paths.
- Added timeout handling to prevent tests from hanging.
- Created a separate smoke test to validate frontend–backend communication.

**2. Use of GitHub Copilot**

**Example Usage:**

GitHub Copilot was used while writing the storage.py and gui.py files. It suggested code completions such as:

- Auto-generating dictionary structures for JSON item creation
- Suggesting regex patterns for date validation
- Completing Tkinter widget definitions
- Proposing HTTP request patterns using requests

**Corrections Made to Copilot Suggestions:**

- Some Copilot-generated code used deprecated Tkinter methods and required updates.
- Several completions lacked proper error handling; custom try/except blocks were added.
- JSON writing suggestions were modified to include indent=2 and ensure_ascii=False.
- Copilot sometimes produced incomplete update logic, which was rewritten manually.

**3.Reflection on AI usage**

The AI tools accelerated development by providing structural templates, helping debug errors, and suggesting improvements in code organization. However, several modifications were required to ensure correctness, alignment with assignment requirements, and adherence to good programming practices. Input validation, modular design, and persistent JSON storage were implemented manually after reviewing AI output. All AI-generated content was reviewed, corrected, and adapted before inclusion in the final project.