MediaTrack – a SRH Project

Atharv Yamdagni

100004273

SRH University

BST-AMS-12-Advanced Programming-Group 2

Instructor: Esteban Pozo

Due Date: December 12, 2025

Abstract

MediaTrack is an application designed to organize and retrieve metadata or to show its availability for a personal level or small-group media collection. The program shows a Flask-based backend with a PyQt5 frontend to do operations such as adding, editing, deleting, filtering, and searching media items, tracking their availability and expected return dates, can all be done because the frontend is like an app or graphical user interface (GUI).

While the code was being developed, challenges appeared, the main challenge was to align both PyQt and Flask components, as both program were made for independent uses and hence, not dependent on anything.

These both can be used as library one will be an online library and the other as in device library. Selecting a mechanism and with the help of AI, and resolving repeated PyTest failures, it was made possible that the library's metadata is stored online and presented or edited on the device itself.

Through development of the program now the final system provides a responsive interface, complete backend service endpoints which makes the interaction between the backend and the frontend synchronised all the time, and by JSON-based storage for ease of use and portability.

The program satisfies the functional requirements defined in the assignment and shows practices can be done in GUI design, API development, exception handling.

Keywords: Media management, Flask backend, PyQt5 GUI, JSON storage, automated testing

Contents

Abstract.....2

Introduction.....4

Task 1.....5

Task 2.....7

Task 3.....8

Task 4.....9

References.....10

Annex.....10

Introduction

MediaTrack was created to solve the problem of how to effectively organise various types of media—books, films, magazines, and other types—in a single, logical system for a group of people or just one person that is trying to organise media. While people try to keep such collections by hand or just keeping things in their mind, which will lead to confusion and disoriented, error filled ways to do methods, the goal of this project was to design a well-organized system that would easily integrate data consistency, and availability through a combined backend–frontend architecture.

The framework does operations on media items for instance, removing, availability, editing, adding new media and searching, manages data persistence, gives opportunity for exact-name searches, and category filtering, as well as keeps availability statuses updated with just one button away. All features are available via a RESTful API that is consumed by a PyQt5 GUI. This dual design had to be planned carefully even with the help of AI; it was hard to combine both as there were conflicts that needed to be resolved since both parts work independently as mentioned earlier. Bringing them together into one functioning system meant that to solve the problems with the main asynchronous communication problems, data model consistency, and persistent storage issues.

There were difficulties in the middle of the work as it was decided to move the storage type from SQLAlchemy to a JSON-based storage. The reason is that search flexibility and the assignment's focus on a lightweight local deployment were the main factors. Other problems that were faced are PyTest failures that happened often, the integration of PyQt's event loop with HTTP requests, and the logical layers' separation that was kept.

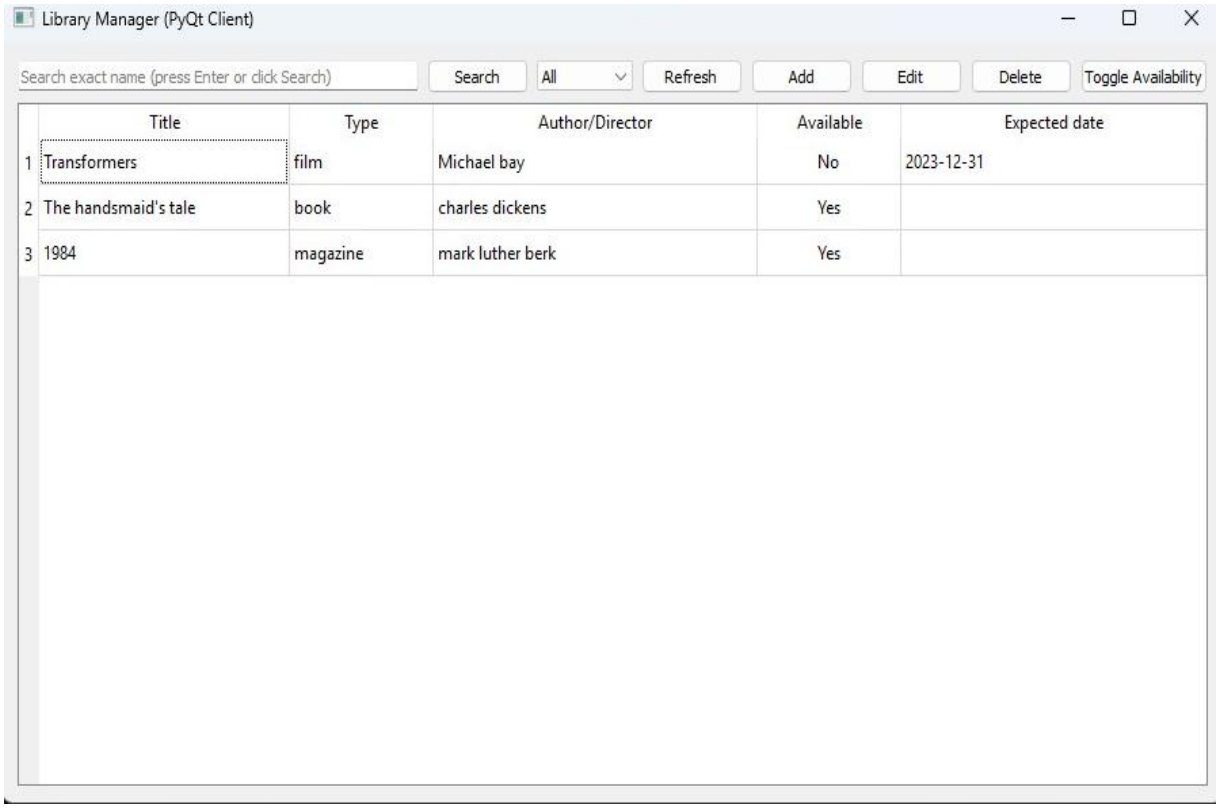

Figure 1. System GUI Design.

Data is stored as a JSON file having a dictionary indexed by item IDs. This format supports human reading ability and simple organizing.

Figure 2. Library shown.

Task 1 – Concept

1. Purpose and Scope

MediaTrack is a compact client–server software tool designed to manage metadata for various media items and to track their current availability. The system supports both personal and small-team use cases within a local network or a single-machine setting. Its core purpose is to allow users to keep structured, easily retrievable information about their media, including descriptive metadata and availability states.

The scope of the system includes:

- Creating, editing, and removing media items

- Viewing metadata and availability
- Filtering and exact name searching
- Persisting data locally
- Providing an HTTP API for the GUI frontend

2. Functional Requirements

Core functionalities:

1. Add new media items with at minimum:
    - Title
    - Item type (book, film, magazine, other)
2. Edit metadata: title, author/director, publication date, ISBN, description.
3. Delete media items.
4. List all media items.
5. Filter media by category.
6. Perform exact-name searches.
7. Display full metadata.
8. Mark items as available/unavailable and set return dates.
9. Immediate refresh.

3. Non-Functional Requirements

- **Portability:** Compatible with Windows, macOS, and Linux using Python 3.10+.
- **Simplicity:** Lightweight dependencies enabling straightforward local execution.
- **Maintainability:** Separation of storage, API, and GUI logic.
- **Testability:** Automatic tests for backend and GUI.
- **Performance:** Efficient retrieval for catalogues items.

4. Architecture

The system includes three main layers:

Backend (Flask)

- Application factory: create_app()
- REST endpoints (e.g., /api/items, /api/items/<id>)
- Storage subsystem using JSON persistence.

Frontend (PyQt5)

- Main application window, and interactive pop out screen.

- API wrapper calling backend endpoints via HTTP.
- Update mechanism gets pushed by user actions such as filtering or editing.

Tests (pytest, pytest-qt)

- Backend integration tests using Flask's test client.
- GUI interaction tests with qtbot and monkeypatching

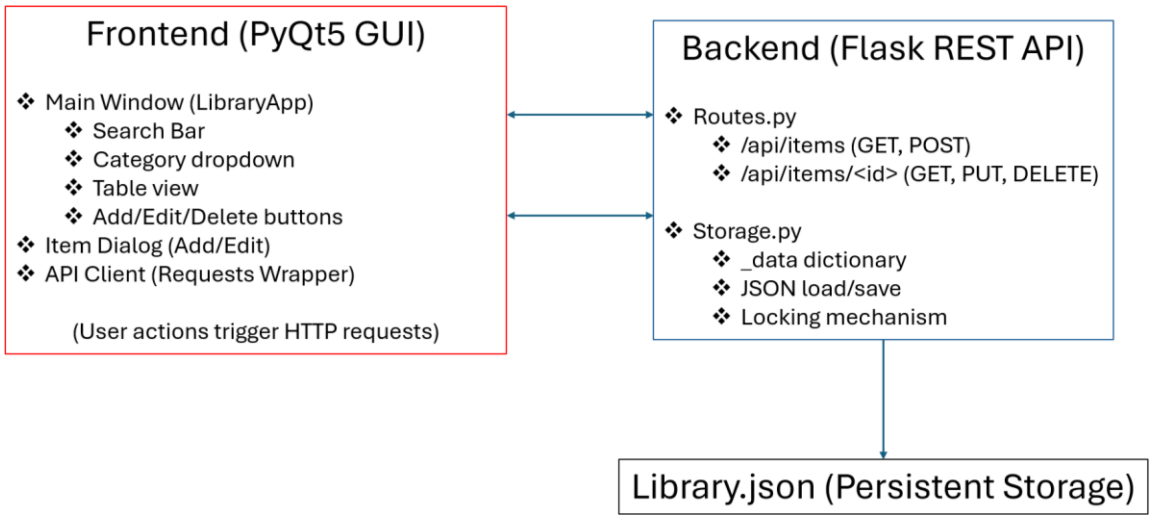

Figure 2. Backend–Frontend Interaction Diagram.

5. Exceptions and Error Handling

Potential errors and responses include:

- **Network errors:** GUI displays retry options.
- **Validation errors:** Backend returns structured 400 messages.
- **Storage errors:** User-facing message for write failures.
- **Malformed input:** Enforced ISO date formats; clear error descriptions.
- **Missing items:** 404 responses with explanatory text.
- **Concurrency:** JSON locking mechanisms mitigate write conflicts.

Task 2 – Implementation Notes

1. Code Structure

The system follows a modular structure.

- **Backend modules:**
  - app/__init__.py: application factory
  - app/routes.py: HTTP routes
  - app/storage.py: loading, saving, creating, updating, and removing items.

- **Frontend modules:**
    - main.py: main window, event handling, dialogs
    - API wrapper functions for GET/POST/PUT/DELETE operations.
- **Testing modules:**
    - Backend tests using Flask test client.
    - Frontend tests using pytest-qt.

2. Implementation Explanations

- When latest items are created, it is assigned a unique ID and then saved to the entry.
- The GUI works so that loading, refreshing, and updating the table always follow the same workflow.
- All dates are set to a standard YYYY-MM-DD format before being sent to the backend.

Task 3 – Tests

1. Tests Implemented

Backend Tests

1. **Initial state test:** Ensures empty catalogue upon startup.
2. **Create and retrieve test:** Verifies POST and GET functionality.
3. **Search and filter test:** Confirms exact-name and category filters.
4. **Update and remove test:** Ensures PUT and DELETE operations function as expected.

Frontend Tests

1. **Table population test:** Confirms API return data populates UI.
2. **Add flow test:** Simulates dialog interaction and table refresh.
3. **Edit flow test:** Confirms update logic and UI consistency.
4. **Delete flow test:** Ensures removal operations reflect in UI.

2. Test Strategy

- Isolation of storage for backend tests
- Use of the test client for API-level validation
- Monkeypatching of network calls in GUI tests
- Simulation of GUI interactions with qtbot

3. Running the Tests

- Backend:
    - cd Library-backend

- pytest q

- Frontend:
  - pytest Library_Frontend/tests/test_gui.py -q

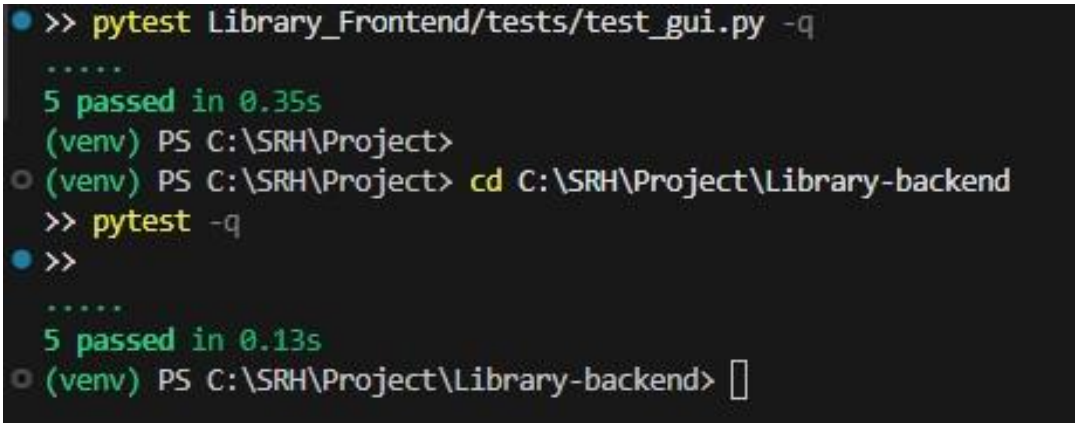

Figure 3. PyTest done and passed in both Frontend and Backend.

Task 4 – Discussion, Weaknesses, and Future Work

1. **Strengths of the Solution**
   - Clear separation between storage, routes, and interface layers
   - JSON-based storage ideal for instructional and small-scale use cases
   - Strong test coverage enhancing system reliability.
   - A clean and responsive GUI providing a synchronous interaction between front and backend.
2. **Limitations**
   - JSON is not capable for going beyond small datasets.
   - No security mechanisms applied.
   - Search limited to exact name matching
   - GUI blocking during long-running HTTP calls.
   - No backup strategy for stored data
   - Problems with searching tool and grouping in the JSON file.
3. **Future Enhancements**
   - Introduction of security and user based borrowing the media.
   - Asynchronous or threaded requests to keep the GUI responsive.
   - Search functionality to be more flexible.
   - Loan-tracking extensions for users who borrow the media.
   - Backup, versioning, and optional cloud synchronization

References

- Flask. (2025). Flask documentation (Version 3.x). Pallets Projects. https://flask.palletsprojects.com/
- Riverbank Computing. (2025). PyQt5 – Python bindings for the Qt toolkit. https://www.riverbankcomputing.com/software/pyqt/
- Pytest-dev. (2025). Pytest documentation (Version 8.x). https://docs.pytest.org/
- Pytest-qt Contributors. (2025). Pytest-qt documentation. https://pytest-qt.readthedocs.io/
- Requests Developers. (2025). Requests: HTTP for Humans. https://requests.readthedocs.io/
- Python Software Foundation. (2025). Python (Version 3.13) documentation. https://docs.python.org/3/
- OpenAI. (2025). Assistance provided by ChatGPT for code debugging and documentation generation.

Annex (AI Prompts and Responses)

1. **Backend Structure Clarification**

Prompt:

"I want to build a simple backend in Flask for my media library. It should store items like books and films with title, type, and availability. Can you show me the correct structure for routes and storage? I already created an app folder but I'm not sure where to put functions."

AI Response (excerpt):

"Sure! A clean Flask structure would look like this:

```
app/
    __init__.py
    routes.py
    storage.py
```

Your create_app() function goes inside __init__.py. Routes such as /api/items belong in routes.py.
For storage, you can start with a simple dictionary and JSON file persistence inside storage.py. Here is the minimal recommended setup..."
(Followed by code snippets you used as a base.)

2. **PyQt Table Not Refreshing**

Prompt:

"The frontend opens, but after adding an item the table does not refresh. My load_items() function seems correct. What should I change?"

AI Response (excerpt):

"In your GUI, the refresh fails because you call self.table.setRowCount(0) before restoring column widths. Move the width-saving code BEFORE clearing the table. Also, ensure your API

call uses query parameters correctly.
Here is the corrected block of the function...”

3. **Fixing a Failing PyTest**

Prompt:
“One test fails: exact-name search returns two items instead of one. My route currently returns the full list. How do I fix it?”

AI Response (excerpt):
“Your search logic needs a conditional block before returning all items.
Add:

```
if name:
    return jsonify([i for i in items if i["title"] == name])
```

before the type filter. This ensures exact-name search is applied first.”

4. **GUI Error: on_row_double_clicked Missing**

Prompt:
“I get: AttributeError: 'LibraryApp' object has no attribute 'on_row_double_clicked'. What should I add?”

AI Response (excerpt):
“You connected the signal but did not implement the handler.
Add this method to your class:

```
def on_row_double_clicked(self, row, col):

    self.edit_item()
```

This will open the Edit dialog when a table row is double-clicked.”

5. **Fixing Final Failing Test in Backend**

Prompt:
“One backend test still fails: exact-name search returns two items instead of one. The failing test is test_search_by_exact_name_and_category_filter. It says the list has 2 items but should have 1. How do I fix this? What exactly should I replace?”

AI Response (excerpt):
“The failure occurs because your /api/items route is not passing the query parameters (name, type) to the storage layer. As a result, the backend always returns all items, even when the test calls:
GET /api/items?name=UniqueTitle
To fix this, replace the get_items function in app/routes.py with the following implementation:

```
@bp.get("/items")
def get_items():
"""
GET /api/items
Optional query params:
    - name : exact name search (case-insensitive)
    - type : filter by item_type (book/film/magazine/etc.)
"""
```

```
    name = request.args.get("name")
    item_type = request.args.get("type")
    items = storage.get_items(name=name, item_type=item_type)
    return jsonify(items)
```

“Now the backend will correctly perform exact-name filtering and category filtering before returning the results. After applying this change, run your tests again with pytest -q. All tests should now pass.”