НАЦИОНАЛНА ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО КОМПЮТЪРНИ ТЕХНОЛОГИИ И СИСТЕМИ - ГРАД ПРАВЕЦ ПРИ ТУ-СОФИЯ

Курсов Проект

ТЕМА: Bookstore Desktop Application

Професия: код 481020 “Системен програмист”
Специалност: код 4810201 “Системно програмиране”

Ученик: Виктор Райков
Курсов номер: 21105

Ръководител:
Красимир Илиев

гр. Правец, 2025 г.

I. Contents

II. Utilised Technologies

The app is a Java-based desktop application, built entirely with Java 23 to take advantage of its latest features, performance improvements, and long-term support. Java was chosen for its platform independence, strong ecosystem, and robust support for enterprise applications.

The application follows a modular Client-Server Architecture, meaning it is divided into separate components: a Swing-based frontend (client) for user interaction, a Spring-based backend (server) that processes requests and handles business logic, and a MySQL database for data storage.

This architecture was chosen for better separation of concerns, making the system easier to maintain, debug, and scale. The frontend focuses solely on the user interface, while the backend manages data processing and security, ensuring a clear division of responsibilities. Additionally, this structure provides scalability and flexibility, allowing future expansions such as integrating a web or mobile frontend without major changes to the backend.

Maven is chosen as the build tool for managing dependencies, structuring the project, and automating the build process. It simplifies integration of libraries like Spring Boot and MySQL connectors, ensures a consistent build workflow, and supports plugins for tasks such as testing and deployment. This improves development efficiency and maintainability.

List of Maven dependencies, used by the whole project:

- Lombok - v. 1.18.36
  - Used for annotations, making the code more readable and less prone to boilerplate.
- ModelMapper - v. 3.2.2
  - Used for mapping an object from one type to another. (eg. Model to Dto)

Backend

The backend is a RESTful API built with Java Spring 3, chosen for its robust architecture, scalability, and ease of integration with other technologies.

Spring provides a well-structured framework for building enterprise-grade applications, offering features such as dependency injection, transaction management, and security support. Its built-in support for RESTful web services makes it an ideal choice for handling client-server communication efficiently.

Additionally, Spring integrates seamlessly with databases through Spring Data JPA, simplifying data persistence and reducing boilerplate code. The framework's modularity allows for better maintainability and scalability, ensuring that the backend can handle increasing workloads and future enhancements without major refactoring.

List of the Maven dependencies, used specifically by the backend:

- Spring Boot Starter Data JPA - v. 3.4.4
  - Starter for using Spring Data JPA with Hibernate.
- Spring Boot Starter Web - v. 3.4.4
  - Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container.
- MySQL Connector/J - v. 9.2.0
  - JDBC driver. Used for communication with a MySQL database.

- Spring Boot Starter Test - v. 3.4.4
    - Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito.

Frontend

The frontend is a desktop GUI application developed in Java Swing, chosen for its flexibility, built-in support in Java, and ability to create cross-platform user interfaces without additional dependencies.

Swing provides a rich set of UI components, allowing for a fully customizable interface that integrates well with the application's requirements. Since Swing is lightweight and runs directly on the Java Virtual Machine (JVM), it ensures consistent performance across different operating systems.

List of Maven dependencies, used specifically by the frontend:

- Jackson Core - v. 2.18.3
    - Used for reading, writing, and streaming JSON data in Java applications

Database

The application uses MySQL 8 as the database because it is reliable, easy to set up, and integrates well with Spring Boot. It efficiently handles structured data, supports SQL queries, and works seamlessly with Spring Data JPA for simplified data access.

MySQL is a good choice for this project because it is widely supported, performs well for most applications, and provides features like indexing, transactions, and scalability when needed.

The user also has a choice to use an In-Memory repository which is implemented without any external modules.

III. Project Structure

The root directory of the project contains the global Maven configuration along with three distinct modules:

- Backend
- Frontend
- Launcher

Launcher

The Launcher module serves as the entry point for the entire application. It contains a single class, ApplicationLauncher, which includes a static main method responsible for orchestrating the startup process.

First, the launcher initializes the backend by invoking its Main method. It then continuously checks the backend's status by sending periodic GET requests to a designated status endpoint. Once the backend is fully operational, the launcher proceeds to start the frontend by calling its Main method. This ensures a smooth and sequential startup process, preventing the frontend from initializing before the backend is ready.

Backend

The Backend module is responsible for handling the application's business logic, data management, and API interactions. It is organized into several key packages to maintain modularity and clarity, making it easier to maintain, extend, and adapt as the application evolves.:

- service
  - Contains the business logic of the application, processing data and coordinating between the repository and controller layers.
- controller
  - Defines the REST API endpoints, handling incoming requests and mapping them to appropriate service-layer methods.
- repository
  - Manages data persistence and retrieval. This package includes a common interface that defines standard data access methods, with one implementation storing data in memory and another integrating with a MySQL database for persistent storage.
- model
  - Contains the core domain entity that represents the application's data structure.
  - dto (Data Transfer Object)
    - A subpackage that defines DTOs for transferring data between layers, ensuring a clear separation between internal models and external representations.
- configuration
  - Holds configuration-related classes, currently including setup for object mapping to simplify data transformations between models and DTOs.

Frontend

The Frontend module is built using Java Swing, hence follows a highly modular design, emphasizing the core OOP principles as well as the Single Responsibility Principle. Due to the complexity of UI management and backend communication, the module is structured into distinct packages that separate concerns, making it easier to manage, extend, and debug.

Each package focuses on a specific aspect of the application's functionality, ensuring clear organization and maintainability.

- model
    - Contains the application's data model, mirroring the structure of the backend to ensure consistency when handling data.
- service
    - Manages interactions with the backend. It includes a service class responsible for sending HTTP requests, processing responses, and providing the necessary data to the UI components.
- gui
    - Contains all graphical user interface components, structured into subpackages based on their roles within the application.
    - main
        - Includes essential UI elements such as the main frame, main panel, top bar, and a Regime enum. The Regime enum determines whether the application is in create, edit, or read mode, dynamically adjusting the UI accordingly.
    - listener
        - Defines listener interfaces to facilitate event-driven communication between UI components. These interfaces enable objects to respond to user actions such as button clicks, book selections, or regime changes.
    - web
        - Houses components that interact with the backend, further divided into:
        - get
            - Contains UI components responsible for retrieving and displaying information when the application is in read mode.
        - post
            - Includes elements used for creating new entries, such as input forms and submission buttons.
        - put
            - Manages components for modifying existing data when the application is in edit mode.

IV. Application operation and features

Backend

The backend API provides a robust and flexible interface for managing a collection of books. It exposes multiple RESTful endpoints, allowing users to perform various operations such as retrieving, adding, updating, and deleting books. The API supports searching for books by ISBN, title, or author, as well as batch operations for creating multiple books at once.

Additionally, the backend offers functionality to switch between different storage options, enabling users to toggle between the MySQL database and an in-memory repository.

A status endpoint is also available to check whether the backend is running. It is used by the application launcher to determine whether the backend has been fully started in order for it to start the frontend.

A book model is made up of four fields:

- isbn. A unique identifier of the Book, it is realized as an UUID.
- name. The name of the book. Its type is String.
- author. The author of the book. Its type is String.
- yearPublished. The book's year of publishing. It is an Integer.

The table below outlines the available endpoints, their HTTP methods, expected request and response formats, and the success status codes returned by the server.

Feature	Endpoint	Method	Success Status	Request body	Response body
Get all of the books in the selected repository.	/books	GET	200	-	[{isbn, name, author, yearPublished }, {...}]
Get book by its isbn	books/{isbn}	GET	200	-	{isbn, name, author, yearPublished }
Get books by their author	books/author/{author}	GET	200	-	[{isbn, name, author, yearPublished }, {...}]
Get books by their title	books/title/{title}	GET	200	-	[{isbn, name, author, yearPublished }, {...}]

Create a new book	/books	POST	201	{name, author, yearPublished }	{isbn, name, author, yearPublished }
Create multiple books in one request	books//bulk	POST	201	[{name, author, yearPublished }, {...}]	[{isbn, name, author, yearPublished }, {...}]
Delete a book by its isbn	/books/{isbn}	DELETE	204	-	-
Update the values of an existing book	/books/{isbn}	PUT	200	{name, author, yearPublished }	{isbn, name, author, yearPublished }
Switch repository between database and in-memory	/books//db/{useDb}	PUT	204	-	-
Get the status of the backend (is it running)	/books/status	GET	200	-	-

Fig. 1

Java classes. Their fields, methods and uses.

***controller*.BookController**

The BookController class serves as the RESTful interface for managing book resources, handling HTTP requests while delegating business logic to BookService.

Annotated with @RestController and mapped to /book, it leverages constructor-based dependency injection via Lombok's @RequiredArgsConstructor.

The controller processes BookDto payloads, ensuring a clear separation between API representations and domain models. Responses are wrapped in ResponseEntity<T> for explicit HTTP status codes.

It also supports dynamic repository switching between an in-memory implementation and a database-backed persistence layer.

All endpoints, along with their request requirements and responses, are documented in Fig. 1, outlining frontend-backend interactions.

Error handling is managed by service-layer validation, and a status endpoint ensures backend availability.

***service*.BookService**

The BookService class encapsulates the business logic for book management, serving as an abstraction layer between the BookController and repository implementations. It is annotated with @Service for Spring's component scanning and uses constructor-based dependency injection (Lombok's @RequiredArgsConstructor).

It supports CRUD operations on Book entities, using ModelMapper to transform BookDto payloads into domain models. Bulk insertion is optimized via Java Streams. The service interacts with the persistence layer through BookRepository, enabling decoupling from specific implementations.

A key feature is repository switching: switchRepository() dynamically toggles between an in-memory (InMemoryBookRepository) and database-backed (DatabaseBookRepository) implementation at runtime. The active repository instance is initialized via @PostConstruct, defaulting to the database repository.

***model*.Book**

The Book class is a JPA entity that represents a book in a database. It is annotated with @Entity, indicating that it maps to a database table. The class uses Lombok annotations @Getter and @Setter to generate get and set methods, @AllArgsConstructor to create a constructor with all fields, and @NoArgsConstructor for a no-argument constructor, which is required by JPA. The isbn field, marked with @Id, serves as the primary key and is of type UUID to ensure uniqueness. The class includes fields for:

- The book's title
- The author's name
- The year of publication

***model.dto*.BookDto**

The BookDto class is a Data Transfer Object (DTO) used exclusively for creating books, not for displaying them. Since the isbn is generated automatically and not provided by the user, this DTO only includes title, author, and yearPublished, ensuring a clean and controlled book creation process.

***repository*.BookRepository**

The BookRepository interface defines methods for managing books in the system. It includes addBook(Book book) and saveBook(Book book) for adding or updating books, removeBook(UUID isbn) for deleting a book by its ISBN, and getBookByIsbn(UUID isbn) for retrieving a book by its unique identifier. It also provides findBookByTitle(String title) and findBookByAuthor(String author) for searching books, along with getAllBooks() to return a list of all books.

***repository*.DatabaseBookRepository**

The DatabaseBookRepository interface is a JPA-based implementation of BookRepository, extending JpaRepository<Book, UUID>. It is annotated with @Repository to indicate that it is a Spring Data repository and @Primary to prioritize it over the In-Memory repository. It provides default implementations for the BookRepository interface, using save(), findById(), deleteById(), and findAll() to handle book persistence. Since JpaRepository already provides built-in methods for most of these operations, this implementation primarily serves as a simple wrapper with default behavior.

***repository*.InMemoryBookRepository**

The InMemoryBookRepository class is a simple in-memory implementation of BookRepository, using a HashMap where the key is the isbn and the value is a Book object

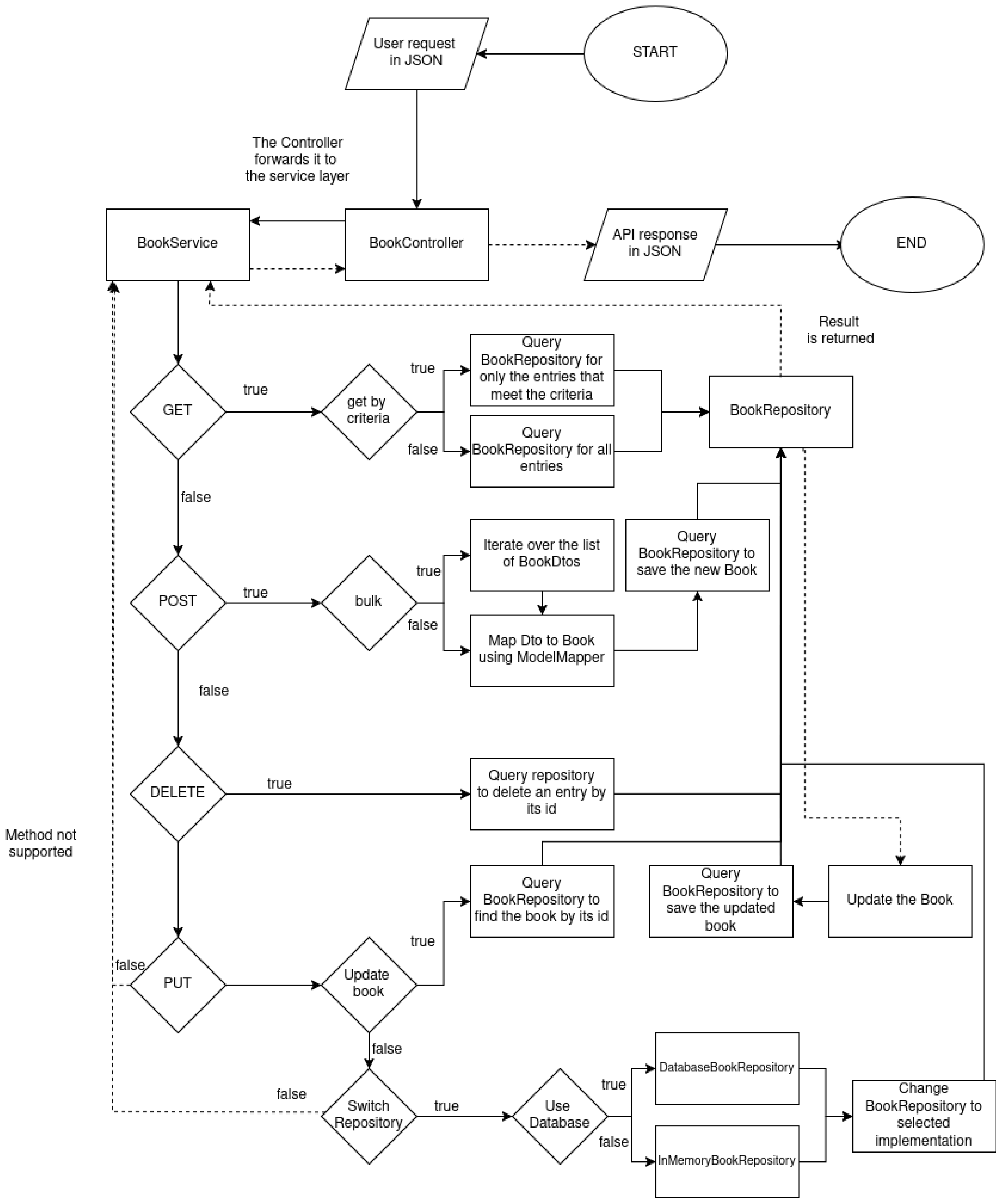
***configuration*.ModelMapperConfig**

The ModelMapperConfig class is a Spring configuration that creates and registers a ModelMapper Bean with a custom converter (DtoToBookConverter) for mapping between DTOs and Book objects. ModelMapper is a library that helps map data between objects, typically between DTOs and entities, to avoid manual copying of properties. The modelMapper instance has to be a Spring Bean so that it can be managed by the Spring container and injected wherever needed in the application.

***configuration.converter*.DtoToBookConverter**

The DtoToBookConverter is a custom ModelMapper converter that implements the Converter interface to convert a BookDto to a Book entity. It generates a new Book object using data from the BookDto and assigns a random UUID to the Book's ID. The @Component annotation makes it a Spring-managed bean, allowing it to be automatically injected into the ModelMapper configuration.

Workflow. Block chart



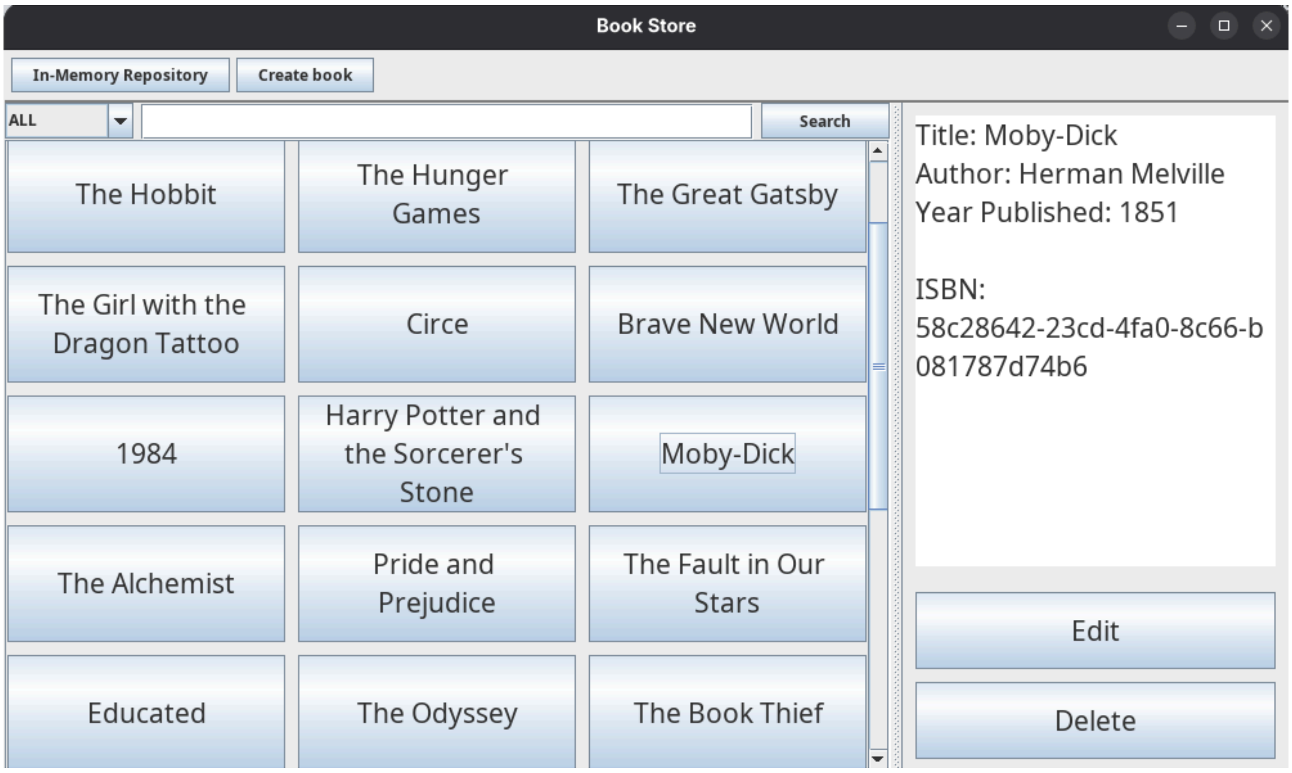
START
User request in JSON
The Controller forwards it to the service layer
BookService
BookController
API response in JSON
END
Result is returned
GET
true
get by criteria
true
false
Query BookRepository for only the entries that meet the criteria
Query BookRepository for all entries
BookRepository
false
POST
true
bulk
true
false
Iterate over the list of BookDtos
Map Dto to Book using ModelMapper
Query BookRepository to save the new Book
false
DELETE
true
Query repository to delete an entry by its id
Method not supported
PUT
false
Update book
true
Query BookRepository to find the book by its id
Query BookRepository to save the updated book
Update the Book
false
Switch Repository
false
true
Use Database
true
false
DatabaseBookRepository
InMemoryBookRepository
Change BookRepository to selected implementation

Fig. 2

Frontend

This Java Swing application provides a graphical interface for interaction with the backend. The frontend communicates with a backend API to fetch, create, edit, and delete books. Users can browse books, search by a specific criteria, switch between different storage modes, and manage book details through an interactive UI.

Image of the application upon startup (Fig. 3):



Title: Moby-Dick
Author: Herman Melville
Year Published: 1851

ISBN:
58c28642-23cd-4fa0-8c66-b081787d74b6

Fig. 3

Functionality

- Book Listing
  - Displays all books retrieved from the backend.
- Search
  - Users can filter books by title or other attributes.
- Book Management
  - Users can create, update, or delete books.
- Storage Mode Switching
  - Allows toggling between an in-memory repository and a database.

UI Component Overview

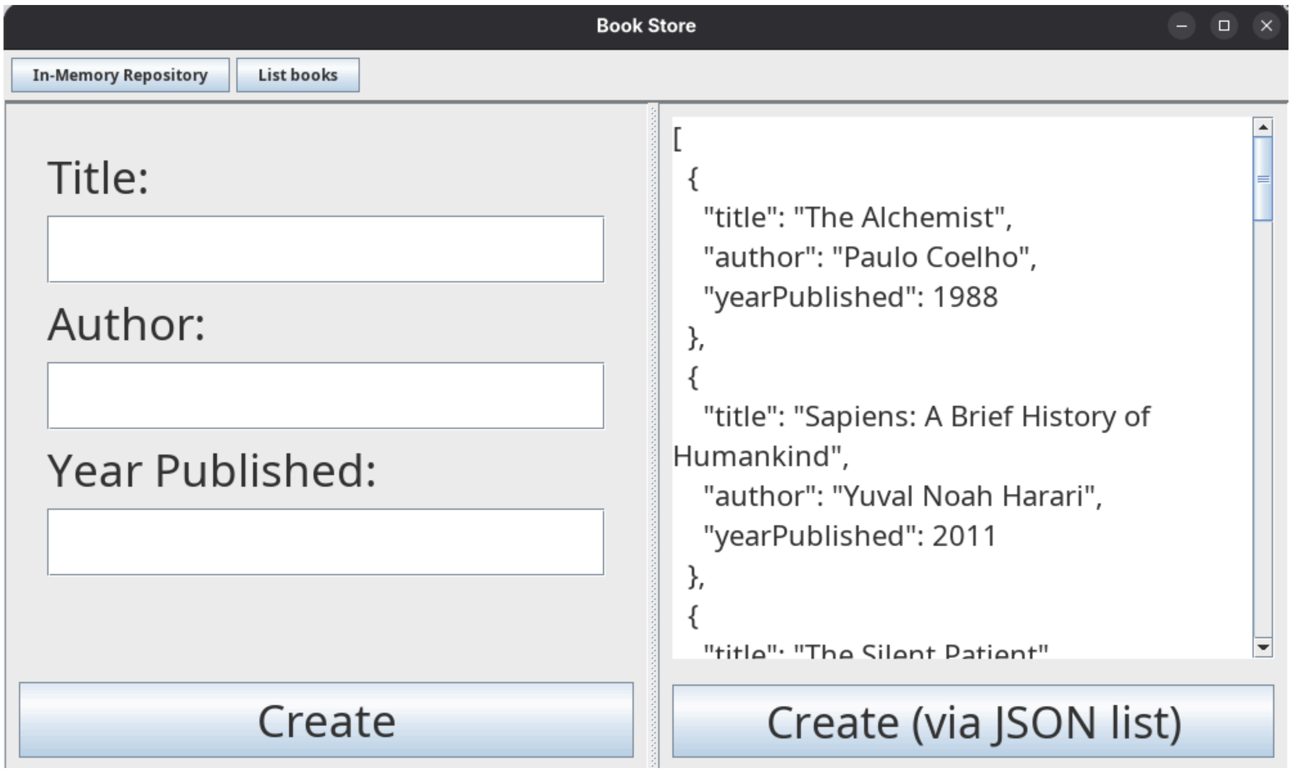
- Book Grid Panel
  - Displays books in a structured button layout.
  - Each button represents a book, and clicking it loads its details.
  - Search panel

        - A text field with a search button to filter books and a criteria dropdown menu that lets the user pick a criteria by which to search.
- Details Panel:
    - Located on the right side, showing selected book information including title, author, year of publication, and ISBN.
    - It includes "Edit" and "Delete" buttons for managing the book.
- Top Panel: Contains controls for:
    - Repository Selection: A button ("In-Memory Repository") to switch between different storage modes.
    - Book Creation: A "Create book" button to open the book creation form.

UI Structure

The UI is structured using a panel-based approach:

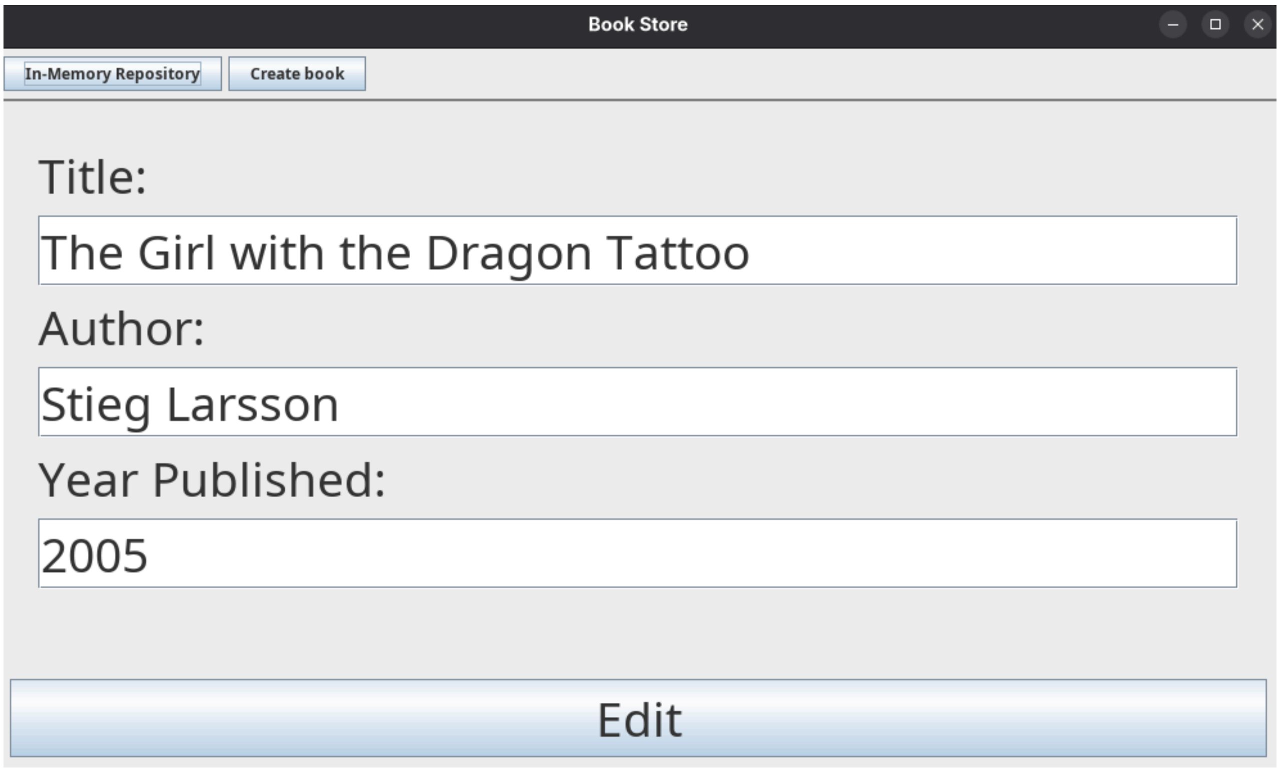
- MainPane
    - The central container that manages different views (book list, create, edit).
- TopPanel
    - Contains repository switching, search, and book creation buttons.
- AllBooksPane
    - Displays all books in a scrollable grid. (Fig. 3)
- CreateBookPane
    - Provides an interface for adding new books. It also provides functionality for bulk book creation using a JSON list. (Fig. 4)



```
[
 {
   "title": "The Alchemist",
   "author": "Paulo Coelho",
   "yearPublished": 1988
 },
 {
   "title": "Sapiens: A Brief History of
Humankind",
   "author": "Yuval Noah Harari",
   "yearPublished": 2011
 },
 {
   "title": "The Silent Patient"
```

Fig. 4

- EditPane – Allows modifying an existing book's details. (Fig. 5)



Title:
The Girl with the Dragon Tattoo
Author:
Stieg Larsson
Year Published:
2005

Fig. 5

Java classes. Their fields, methods and uses.

***model*.Book**

The Book class represents a complete book entity, including a UUID as the ISBN, a title, an author, and a publication year. It mimics the backend book model and is used for retrieving and displaying book data from the backend. The toString() method provides a formatted representation of the book's details.

***model*.BookDto**

The BookDto class is a simplified version used for book creation. It excludes the UUID field since the backend generates it automatically. This ensures that users only provide relevant book details when adding entries.

***service*.BookService**

The BookService class acts as the frontend's interface for communicating with the backend REST API. It follows a singleton pattern to ensure a single instance manages all HTTP requests.

The class defines several endpoint URLs and utilizes HttpClient for making requests. It provides all operations supported by the backend. Exception handling is incorporated to catch API errors and ensure smooth execution.

The class also utilizes ObjectMapper and ModelMapper for JSON serialization and object transformations when sending and receiving book data.

***service*.SearchCriteria (Enum)**

The SearchCriteria enum defines different search filters available for retrieving books. It includes ALL, ISBN, TITLE, and AUTHOR, each mapping to a specific API endpoint.

These criteria allow the frontend to query books based on different attributes by appending the appropriate endpoint to the base URL.

***gui.main*.MainFrame (JFrame)**

The MainFrame class serves as the main application window. It initializes the GUI by setting up the frame with a title, a predefined size of 1000x600 pixels, and a default close operation to exit when the window is closed.

The class adds an instance of MainPanel as the primary content container and makes the frame visible to the user. This class acts as the entry point for launching the frontend.

***gui.main*.MainPanel (JPanel)**

The MainPanel class acts as a wrapper for MainPane and is responsible for handling layout adjustments. It extends JPanel and contains a MainPane instance, which is added to the center of the layout.

This class ensures that the primary UI components are displayed properly and provides layout management by dynamically adjusting the divider position in its overridden doLayout() method. This ensures that the interface remains proportionally balanced when resized, keeping the book listing and detail panel well-distributed within the frame.

***gui.main*.MainPane (JSplitPane)**

The MainPane class is the central container for managing the primary content of the application. It extends JSplitPane and dynamically switches between three different UI states: listing books, creating new entries, and editing existing ones.

This class integrates multiple components, including TopPanel, which provides controls for repository switching and book management actions, AllBooksPane, which displays the book collection, CreateBookPane, which provides a form for adding new books, and EditPane, which allows users to modify book details.

At its core, MainPane maintains a Regime field that dictates the current UI state. The setRegime(Regime regime) method updates this state and triggers a refresh of the displayed component. To ensure seamless transitions between views, updatePanel() first clears all existing components from the layout before adding the appropriate panel based on the current regime. The getPanel() method determines which component should be displayed by leveraging a switch expression, returning either the AllBooksPane, CreateBookPane, or EditPane as needed.

When switching to the list mode, refreshAndGetAllBooksPanel() ensures that the book collection view is refreshed before being displayed.

The class also establishes communication between components through event listeners. The setListeners() method binds various user interactions to their respective actions. For example, it links TopPanel's repository switch button to the switchRepo() method in AllBooksPane, ensuring that book data is retrieved from the selected storage mode. Additionally, it connects the book editing functionality by allowing AllBooksPane to trigger editBook(UUID id), which sets the EditPane to display the selected book's details before switching the UI to edit mode. The editBookPanel itself can also notify MainPane when editing is complete, reverting the regime back to listing mode.

To maintain UI consistency, MainPane ensures that all updates are immediately reflected by calling revalidate() and repaint() after modifying the layout. These methods guarantee that Swing properly refreshes the components to reflect any changes. The combination of state management, event-driven interactions, and real-time UI updates makes MainPane the backbone of the application's dynamic interface.

gui.main.**Regime (Enum)**

The Regime enum defines the different states the application can be in. It is used in MainPane to determine which panel should be displayed based on user actions. It has three possible values:

- LIST
  - Represents the default view showing all books;
- CREATE
  - Activates the book creation form
- EDIT
  - Allows modification of an existing book.

gui.main.**TopPanel (JPanel)**

The TopPanel class provides user controls for managing the repository and switching between book listing and creation modes. It extends JPanel and is designed with a flow layout to align its components neatly.

It includes a button for toggling between an in-memory repository and a database repository. When pressed, the button updates its label to reflect the active storage mode and triggers an event via RepoSwitchListener.

Another button is responsible for toggling between listing books and opening the book creation form. This button updates its label accordingly and uses RegimeListener to notify MainPane of the state change. The TopPanel ensures that these controls remain accessible at the top of the interface and properly interact with the rest of the application.

***gui.listener.*ActionListener**

This interface defines a contract for performing actions on a selected book using its UUID. It is primarily used in buttons for deleting, editing or creating a new book.

Used in:

- MainPane to provide the edit functionality in BookInfoPanel's ButtonPanel.
- AllBooksPanel to provide the delete functionality in BookInfoPanel's ButtonPanel.

***gui.listener.*BookSelectListener**

Allows components to respond when a book is selected. It provides a method to handle the selected Book object, making it useful for triggering actions like opening a book's details in the editor.

- AllBooksPane to notify BookInfoPanel about the book selected inside GridPanel.

***gui.listener.*RegimeListener**

Enables switching between different UI states (LIST, CREATE, or EDIT).

Used in:

- MainPane to provide toggling list/create in TopPanel.
- MainPane to provide enable switching to edit mode inside EditPanel

***gui.listener.*RepoSwitchListener**

Used for toggling between the storage options.

Used in:

- MainPane to provide repository switching in TopPanel

***gui.listener.*SearchListener**

Facilitates book searches by defining a method that accepts a query string and a SearchCriteria filter. It ensures that components handling searches can request book data based on specific attributes (ISBN, title, or author).

Used in:

- BooksPanel to get notified about new searches made inside SearchPanel.

***gui.web.get.all.*AllBooksPane**

The AllBooksPane class extends JSplitPane and acts as the primary container for managing the book list and its details. It consists of a BooksPanel on the left and a BookInfoPanel on the right, with a fixed resize weight of 2/3 to prioritize the book list.

The class maintains event listeners to synchronize selection and deletion operations between these panels. The setListeners() method binds BooksPanel's selection event to update the *gui.web.get.info.*BookInfoPanel, while also linking book deletion to remove the selected book from the repository.

The switchRepo(boolean useDb) method updates the repository source in BooksPanel, and the refresh() method clears and reloads the displayed books.

***gui.web.get.all.*BooksPanel**

The BooksPanel class extends JPanel and acts as a container for displaying book search results. It initializes a SearchPanel for filtering books and a ScrollPane for listing them.

The setSelectListener(BookSelectListener listener) method binds book selection events, while deleteBook(UUID isbn), switchRepo(boolean useDatabase), and refresh() propagate changes to the ScrollPane.

The panel updates dynamically when a book is added, removed, or searched.

***gui.web.get.all.*SearchPanel**

The SearchPanel class extends JPanel and provides a user interface for searching books based on various criteria. It contains a JTextField for entering search queries, a JComboBox<SearchCriteria> for selecting the search category (all, title, author, or ISBN), and a JButton to trigger the search operation.

The performSearch() method is called when the search button is clicked, passing the trimmed input from searchField and the selected search criteria to a SearchListener. The layout is managed using a GroupLayout, ensuring that components are aligned properly with minimal spacing adjustments.

The configureComponentSize() method standardizes component dimensions, preventing layout inconsistencies. When the search criteria change, the search field is cleared to maintain input relevance.

***gui.web.get.all.*ScrollPane**

The ScrollPane class extends JScrollPane and manages the book display. It fetches books usingBookService.fetchBooks(String query, SearchCriteria searchCriteria), processes search queries, and updates the GridPanel. It acts as a wrapper around GridPanel, providing it with scrolling functionality and providing backend interaction functionality to ensure GridPanel only manages the displaying of the book buttons.

The setSelectListener(BookSelectListener listener) method links book selection events.

deleteBook(UUID isbn), switchRepo(boolean useDatabase), and refresh() ensure data consistency by modifying the backend using BookService and updating the UI. The configure() method sets up scroll behavior, enforcing vertical scrolling while disabling horizontal scrolling.

***gui.web.get.all.*GridPanel**

The GridPanel class extends JPanel and uses a GridLayout to organize book buttons. It dynamically adjusts row count based on the number of books displayed, ensuring a uniform appearance.

When a book button is clicked, it triggers a BookSelectListener, notifying *gui.web.get.info.*BookInfoPanel about the selection.

The displayBooks(List<Book> books) method removes old entries, populates new book buttons, recalculates the grid dimensions, and updates the UI using revalidate() and repaint().

***gui.web.get.all.*BookButton**

The BookButton class extends JButton and provides a UI component for representing books. It applies HTML formatting to center the text and uses a ComponentAdapter to dynamically adjust the font size when resized. The font size is recalculated based on button dimensions, ensuring readability regardless of scaling.

***gui.web.get.info.*BookInfoPanel**

The BookInfoPanel class extends JPanel and serves as a detailed book information display. It consists of two main components: InfoPane, which shows book details, and ButtonPanel, which provides "Edit" and "Delete" actions.

***gui.web.get.info.*InfoPane**

The InfoPane class extends JScrollPane and contains a JTextArea for displaying book information. It supports text wrapping, ensures content is non-editable, and dynamically adjusts font size based on panel dimensions. The setText() method updates the displayed book details.

The class includes methods for setting action listeners for editing and deleting books. It also listens for component resize events to dynamically adjust font sizes for better readability.

***gui.web.get.info.*ButtonPanel**

The ButtonPanel class extends JPanel and organizes two ActionButton instances for editing and deleting books. It arranges them in a vertical grid layout and provides methods for setting the selected book's UUID and assigning action listeners. Font size adjustment is supported for better UI scaling.

***gui.web.get.info.*ActionButton**

The ActionButton class extends JButton and represents an interactive button for performing actions on books (either edit or delete). Each button is associated with a UUID representing the selected book. When clicked, it triggers the corresponding action via its ActionListener.

***gui.web.post.*CreateBookPane**

CreateBookPane extends JSplitPane and serves as a container that splits its layout between CreateSinglePanel and CreateBulkPanel. It does not manage book creation itself but delegates the responsibility to these two child panels.

A ComponentAdapter listens for resize events and dynamically adjusts the divider position to maintain a balanced layout. By using composition, it keeps its responsibilities limited to layout management. (Fig. 4)

***gui.web.post.*CreateSinglePanel**

CreateSinglePanel extends JPanel and provides a structured form-based UI for entering book details. It contains an instance of BookFormPanel, which includes text fields for title, author, and publication year.

It holds an instance of ButtonPanel, which encapsulates a single button labeled “Create.” The button’s behavior is not hardcoded inside ButtonPanel; instead, CreateSinglePanel configures its action by setting an ActionListener that retrieves form data, constructs a BookDto, and sends it to BookService for book creation.

A ComponentAdapter listens for size changes and adjusts font sizes dynamically, ensuring readability.

***gui.web.post.*BookFormPanel**

A JPanel containing three labeled text fields (titleField, authorField, yearField) for book data input. It uses GroupLayout for structured alignment and enforces consistent spacing. Provides getters for retrieving user input. The getYearPublished() method parses the year as an integer.

***gui.web.post.*CreateBulkPanel**

CreateBulkPanel extends JPanel and provides an interface for bulk book creation using a JTextArea. This allows users to input multiple books in JSON format.

Like CreateSinglePanel, it also contains an instance of ButtonPanel, but here the button is labeled “Create (via JSON list).” Instead of handling individual form inputs, the button’s configured action reads the JSON from the text area and sends it to BookService for bulk book creation.

A scrollable text area ensures usability for large inputs, and a component listener dynamically resizes fonts for readability.

***gui.web.post.*ButtonPanel**

ButtonPanel is a reusable component that encapsulates a single button. It is an abstract class that is instantiated inside CreateSinglePanel and CreateBulkPanel as an anonymous class, allowing them to define its behavior independently.

It provides a method setFontSize(int size), enabling dynamic font scaling. The button's action is externally defined using an ActionListener, making it flexible for different use cases. This approach follows the Single Responsibility Principle by keeping ButtonPanel solely focused on rendering and styling the button, while the panels using it define the button's function.

***gui.web.post.*EditPanel**

EditPanel extends CreateSinglePanel, inheriting its structure but modifying it for editing books instead of creating them. This avoids redundant code while maintaining a consistent UI.

Instead of starting with empty fields, EditPanel introduces setBookId(UUID bookId), a method that retrieves a book from BookService and fills the inherited BookFormPanel with its details. This allows the panel to dynamically update based on the selected book, ensuring that users can modify existing information rather than inputting everything from scratch.

EditPanel overrides getButtonPanel() to return a ButtonPanel with an "Edit" button. This button calls editBook(), updating the book's properties and saving the changes via bookService.editBook(book).

By retrieving BookFormPanel from CreateSinglePanel instead of instantiating it, EditPanel maintains encapsulation and ensures consistency. The RegimeListener enables smooth UI transitions without tightly coupling EditPanel to the overall application flow.

Inheritance allows EditPanel to reuse the form layout and logic from CreateSinglePanel, reducing redundancy. At the same time, method overriding customizes behavior where necessary, such as replacing the button panel with an edit-specific implementation. Encapsulation ensures that EditPanel interacts with BookFormPanel without creating unnecessary dependencies.

Workflow. Block chart.

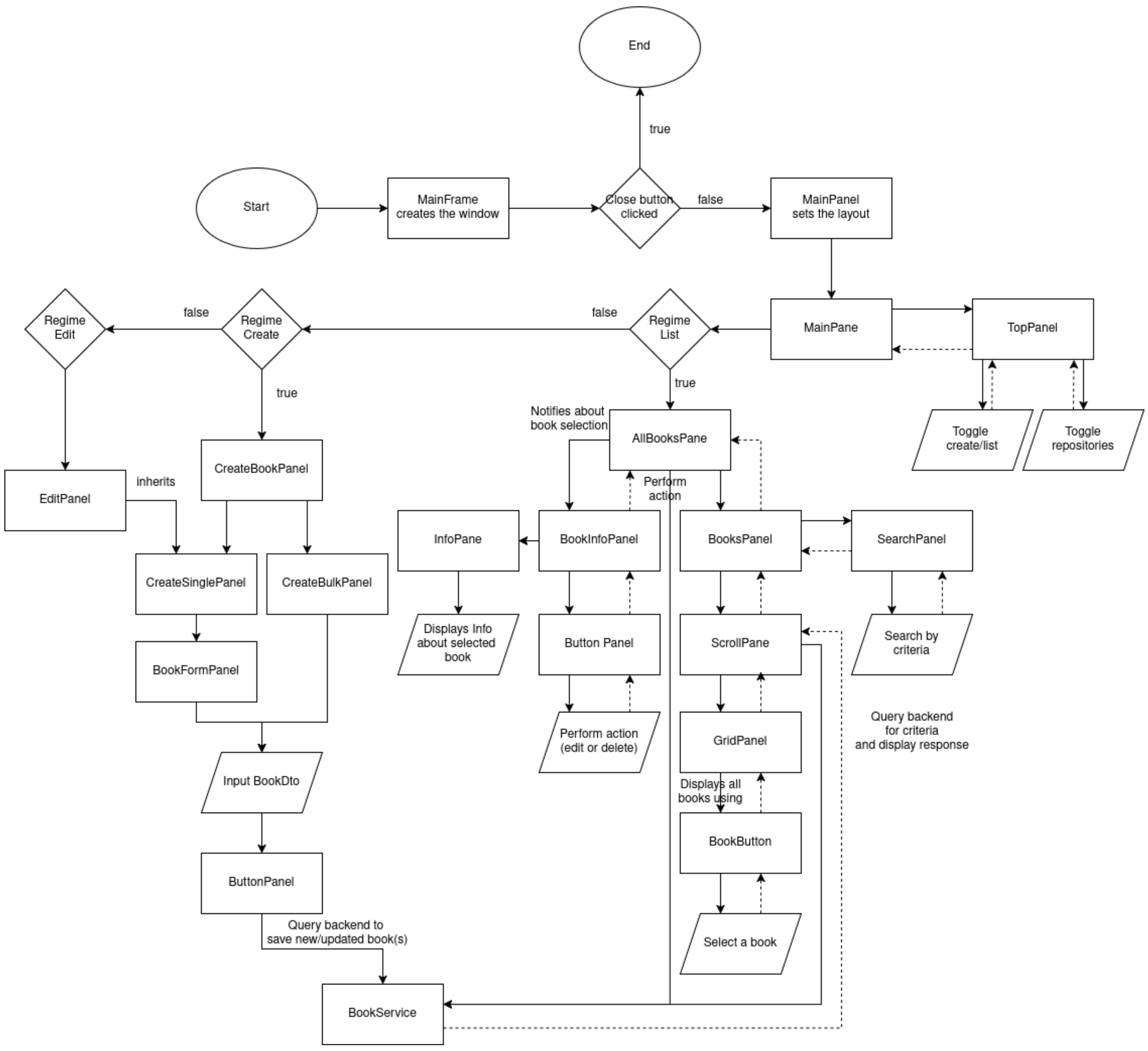


Fig. 6

V. User guide

Viewing and Searching Books

At the top of the interface, you'll find the search bar—a central tool for quickly locating books in the repository. It's made up of two main components: a dropdown menu for selecting the search criteria (Title, Author, ISBN, or All) and a text field where you enter your query. By default, the dropdown is set to "All," which means the app will display every book currently in the system without needing a search term.

Once you type a query and click the "Search" button, the app filters the books based on your chosen criteria and shows the first matching result. If no term is entered and "All" is selected, it will simply list all available books.

Search results appear in a scrollable list, each entry showing the book's title. Clicking on any of these entries opens a detailed view on the right side of the screen. This panel displays the book's full title, author, and year of publication in a larger font, giving you a clear and readable view of the selected book's information.

Editing and Deleting Books

Below the detailed book view, you'll find two buttons: "Edit" and "Delete." If you click "Delete," the book will be removed from the system immediately. Be cautious—there's no pop-up confirmation, so be sure you want to delete it.

Clicking "Edit" will open an editable form with the book's existing information already filled in. You can update the title, author, or year as needed. Once you make your changes, click the "Edit" button again to save. After editing, the app switches back to the list view automatically.

Top Panel

The top panel lets you switch between different parts of the app. You can toggle between *List Mode* (for browsing and managing books) and *Create Mode* (for adding new books, either one at a time or in bulk). There's also a button to toggle between storage options. Changes apply instantly, helping you work the way you prefer without restarting or losing progress.

Adding a Single Book

When in Create Mode the left half of the main screen is dedicated to adding a new book one at a time. You'll see a form with three fields: Title, Author, and Year Published. Enter the details of the book you want to add, then click the "Create" button below. That's it — the new book will be added to your collection and ready to search or edit like any other. Now you can press "List books" in the top panel to show all of the books.

Adding Multiple Books at Once

On the right-hand side of the screen is the bulk creation panel. This is ideal if you already have a list of books and want to enter them quickly. The text area accepts a JSON list format, which allows you to paste in multiple book entries at once. Once your data is ready, click the "Create (via JSON list)" button at the bottom to upload all of them together.

Changing the Repository Type

The application supports switching between different types of data storage - In-Memory and Database. This gives you control over whether the data persists between sessions or resets when you close the app.

To change the repository type, look for a button available at the top of the screen. Once you choose a different repository type, the app may reset its current data view to reflect the new storage source. For example, switching from memory to DB mode will load any saved books, while switching back to memory will give you a clean slate.

Changing the repository type is useful when you want to test data quickly without saving it permanently, or when you're ready to store your collection more reliably between sessions.

Dynamic Interface

The app automatically adjusts its font sizes and layouts based on the size of your window. This means that no matter the screen size or how you resize the application, the text and buttons will remain clear and legible. This ensures a smooth and consistent experience, whether you're working on a small laptop or a large monitor.

VI. Used literature

No external literature was used during the development of this application. All solutions are based on original implementation and existing knowledge.

VII. List of Figures