Ø=ÜÍ Project Report: VITyarthi Student Resource Manager

1. Cover Page

Aryaman Dev

Roll no. 23BAS10087

Date of submission 25th november 2025

2. Introduction

The VITyarthi Student Resource Manager is a console-based application designed to help students organize and track their diverse collection of learning materials (books, videos, notes, links). The project serves as an exercise in applying foundational Java concepts, including Object-Oriented Programming (OOP), modular design, basic data structures (ArrayList), and exception-safe input handling. It provides a structured way to manage resources, moving beyond chaotic file systems or bookmarks.

3. Problem Statement

Students often struggle with managing a high volume of scattered study materials. This results in inefficiency when retrieving specific resources and a lack of insight into the distribution of their learning sources (e.g., how many resources are links versus books). The problem addressed is the need for a centralized, simple, and functional system for tracking academic resources.

4. Functional Requirements (FRs)

The system must be able to perform the following actions:

FR1: User Authentication: Allow a user to Register and Log In to secure the system.

FR2: Resource Creation (C): Allow the user to Add a new resource, specifying a Title, Type (Book/Video/Link/Notes), and Description.

FR3: Resource Reading (R): Allow the user to View a numbered list of all currently stored resources.

FR4: Resource Deletion (D): Allow the user to Delete a resource by providing its index number from the list.

FR5: Report Generation: Generate an analytical report showing the total count of resources and the count for each resource type.

5. Non-Functional Requirements (NFRs)

Requirement Description Strategy in Implementation
NFR1: Usability The application must have a clear, easy-to-navigate CLI menu structure. Main.java implements a simple, nested menu loop (main and studentMenu).
NFR2: Error Handling The system must handle invalid input gracefully without crashing. InputUtil.java is used to manage input (e.g., preventing InputMismatchException for integer entry, though only basic Integer.parseInt is used).
NFR3: Maintainability The codebase must be modular with clear separation of concerns. Implemented using dedicated packages (services, models, utils) and service classes (UserService, ResourceService).
NFR4: Performance Core operations (Add, View, Delete) on resources must be instantaneous for a typical number of resources. Using the standard Java ArrayList provides $O(1)$ average-case performance for adding and $O(1)$ for indexed reading/deletion.

6. Ø<ßSystem Architecture

The application is built on a Three-Layer Architecture (Presentation, Business Logic, and Data) to maintain a clear separation of responsibilities.

Presentation Layer: Handled by Main.java, which manages the command-line menu and user input/output flow.

Business Logic/Service Layer: Handled by the classes in com.vityarthi.services (UserService, ResourceService, ReportService). These classes contain all the rules and logic for managing data.

Data/Model Layer: Handled by the classes in com.vityarthi.models (Student.java, Resource.java), which define the structure and state of the application's data objects.

7. Design Diagrams

7.1. Use Case Diagram

The primary actor is the Student, who interacts with the system through three main use cases: Authentication, Resource Management, and Reporting.

7.2. Workflow Diagram

This diagram outlines the flow of control starting from the main application loop.

7.3. Class Diagram

This diagram details the static structure of the codebase, illustrating class relationships and dependencies.

8. Design Decisions & Rationale

Decision	Rationale
Modular Service Classes	Adheres to the Single Responsibility Principle (SRP). By having separate UserService, ResourceService, and ReportService, each class is focused on a single domain, making the system easier to test, debug, and expand.
Input Handling Utility (InputUtil)	Centralizes all input operations, wrapping the Scanner object. This isolates system resources and provides a cleaner interface for getting validated strings and integers across the entire application.
In-Memory Storage (ArrayList)	Chosen for simplicity and compliance with basic project requirements. It demonstrates the use of core Java data structures. The trade-off is the lack of data persistence.
String Comparison in Report Service	Uses r.getType().toLowerCase() in ReportService when counting resources. This ensures reliability in counting, regardless of whether the user inputs "book," "Book," or "BOOK."
Indexing for Deletion (index - 1)	The UI uses 1-based indexing for user convenience, but the core Java ArrayList.remove() uses 0-based indexing. The conversion (index - 1) is handled in ResourceService for better usability.

9. Implementation Details

A. Core Data Structures

ArrayList<Resource> resources: Used in ResourceService to store the collection of all resources. This allows dynamic sizing and efficient iteration.

Student registeredStudent: Used in UserService to store the single registered user's details.

B. Key Methods and Logic

UserService.login(): Performs a direct string comparison (equals()) between the input credentials and the stored registeredStudent credentials.

ResourceService.addResource(): Creates a new Resource object and appends it to the resources list.

ResourceService.deleteResource(): Calls viewResources() first to guide the user, accepts an integer input, validates the input range, and removes the element at index - 1.

ReportService.generateReport(): Iterates over the entire resources list and uses a switch statement on the lowercase resource type to count the occurrences of each category (book, video, link, notes).

10. Screenshots / Results

(Screenshots demonstrating a full user session would be placed here, including:)

Main Menu and Successful Registration.

Login Screen and Successful Login.

Student Menu and adding 2-3 different resource types.

Viewing Resources (output of viewResources()).

Generating the Report (output of generateReport()).

Successful Resource Deletion.

11. Testing Approach

The testing approach was Manual Functional Testing via the command-line interface.

Positive Testing:

Successfully registering a user with a valid username.

Logging in with correct credentials.

Adding resources of all four types (Book, Video, Link, Notes).

Deleting a resource by entering a valid index.

Verifying the generateReport totals match the number of added resources.

Negative Testing & Edge Cases:

Attempting to register with an invalid username (length < 3).

Attempting to log in with incorrect password/username.

Attempting to delete a resource using an invalid index (e.g., 0, a number greater than the list size).

Generating a report when the resources list is empty.

12. Challenges Faced

The main challenges were in managing the command-line input:

Scanner Handling: The mix of Scanner.nextLine() (used in InputUtil.getString()) and Scanner.nextInt() (used for resource deletion and menu choices) can lead to the Scanner token problem (skipping input). This was mitigated by standardizing all inputs to be read as

strings and then parsing integers using Integer.parseInt(sc.nextLine()) within InputUtil.getInt().

Error Reporting: Providing informative error messages (e.g., "Invalid resource number!") was essential to ensure the user understood why an operation failed, which required careful placement of input validation checks.

13. Learnings & Key Takeaways

OOP Fundamentals: Solidified understanding of encapsulation (e.g., private fields in Resource.java) and modularity through service classes.

Input Handling Robustness: Learned best practices for handling Scanner in CLI applications to prevent runtime errors.

Separation of Concerns: Clearly understanding the roles of the Model, Service, and Main classes to build a well-organized system structure.

14. Future Enhancements

The project can be significantly improved with the following additions:

Data Persistence: Implement storage using File I/O (Serialization) or connect to a SQL database (JDBC) so that resource data persists after the application is closed.

Advanced Resource Attributes: Add more attributes to the Resource class, such as URL link, Course Name, Priority Level, and Completion Status.

Search and Filter: Implement a ResourceService.searchResources(String keyword) method to allow filtering by title or type.

Robust Input Validation: Add checks for resource type to ensure only the allowed types (Book/Video/Link/Notes) are accepted.

15. References

Official Java Documentation

Core Java textbooks and online tutorials for data structures and OOP concepts.