**Development and Demonstration of a Rest API**

**(Library Management Rest API)**

A black and gold logo

Description automatically generated

Submission Date: 24/02/2025

Lecturer Name: Mary Giblin

**Student**

Philip Herweling

A00326153

A00326153@student.tus.ie

AL\_KCNCM\_9

Contents

[**1.** **Introduction:** 3](#_Toc190946026)

[1.1 Overview of the Task: 3](#_Toc190946027)

[1.2 Objectives of the RESTFUL API project 3](#_Toc190946028)

[**2.** **Design and Implementation:** 4](#_Toc190946029)

[2.1 Description of the Rest API: 4](#_Toc190946030)

[2.1.1One to Many Relationships: 4](#_Toc190946031)

[2.1.2 Using Date Objects: 4](#_Toc190946032)

[2.1.3 Data Transfer Objects: 4](#_Toc190946033)

[2.1.4 Database Design: 4](#_Toc190946034)

[2.1.5 Error Handling: 5](#_Toc190946035)

[2.1.6 HATEOAS: 5](#_Toc190946036)

[2.1.7 Final Thoughts: 5](#_Toc190946037)

[**3.** **Code Explanation:** 5](#_Toc190946038)

[**4.** **Challenges and Solutions:** 5](#_Toc190946039)

[**5.** **Conclusion:** 5](#_Toc190946040)

# **Introduction:**

## Overview of the Task:

The purpose of this Assignment was to develop a RESTful API. I choose to build a RESTful API for a Library Management System using Spring Boot as was required. The Library Management system is designed to manage users, books and loans as efficiently as possible by providing structured endpoints which facilitate CRUD operations. The API I built ensures data security and scalability while adhering to the best practices laid out in the brief such as pagination, validation, error handling and HATEOAS principles.

## Objectives of the RESTFUL API project

The primary goal of this Library Management System are as follows:

* User Management: Allow users to be created, updated, retrieved and deleted.
* Book Management: Maintain a inventory of books with all there details, such as there book ID, book title, author, ISBN and Number of copies available/ there availability status
* Loan Management: This allows Users to borrow books, return books and also tracks the due date of loans.
* Validation and Error Handling: The API should have robust error handling and data validation to ensure the reliability of the API.
* Pagination and Filtering: The API should have pagination built in for large datasets and filtering based on due dates. This way the books which are due the soonest can be retrieved with a simple call to an endpoint.
* HATEOAS Implementation: The API should enhance discoverability by adding hypermedia links to the API responses by using HATEOAS.
* Database Integration: The API should use H2 database for seamless storage and retrieval of records i.e. user, book and loan records.

The Library Management System is designed to streamline the library’s operation, cutting down on manual record keeping, and also providing a scalable solution for book and loan management.

# **Design and Implementation:**

## Description of the Rest API:

### 2.1.1One to Many Relationships:

In this RESTful API, I implemented to One to Many relationships, which model real world interactions between entities. The two one to many relationships I implemented are between Users and Loans, and books and loans.

**User 1-----\* Loans:**

A User can borrow multiple books through loan records. Each loan record links a user to a book and tracks the borrowed date and the due date. I achieved this by using JPA @OneToMany and @ManyToOne annotations.

### 2.1.2 Using Date Objects:

For this project is was essential that I handle date inputs and outputs properly for managing loan durations.

**How Dates are Used in this API:**

When creating a loan, the API assigns borrowedDate = LocalDate.now() and dueDate = LocalDate.now().PlusDays(21). The API follows the ISO 8601 format (YYYY-MM-DD) like it was specified on the brief. This keeps the dates consistent. A user can also filter by date using an endpoint GET loans by due date.

### 2.1.3 Data Transfer Objects:

Data Transfer Objects can be used to simplify API response by preventing unnecessary data exposure, formatting and structuring responses effectively and improving maintainability and security.

For this project I made a DTO for all three entities, loan, user and book entities. This allowed me to expose only the necessary fields and return only what the client would need to see.

## 2.2 Database Design:

The Library Management System API uses a H2 Database for development and testing. The database schema consists of three tables, The user table, Book table and the Loan table.

A screenshot of a computer

AI-generated content may be incorrect.

### 2.2.1 Entity Relationship Diagram:

A yellow rectangular sign with black text

AI-generated content may be incorrect.

## 2.3 Error Handling:

In this API I implemented error handling using a GlobalExceptionHandler class which manages three different exceptions and errors:

1. **Validation Error:** This is triggered if an input field is left blank or for example if an email entered isn’t the correct format.
2. **Resource Not Found:** This exception is triggered when an entity can’t be found by there ID for example.
3. **Entity Already Exists:** This exception is triggered if when creating a new user for example an email is used which has already been used by an existing user. This stops one email being used for multiple user accounts.

A screenshot of a computer program

AI-generated content may be incorrect.

## 2.4 HATEOAS:

As mentioned in the introduction HATEOAS enhances API discoverability by adding links to other endpoints in the response from another endpoint call.

A screen shot of a computer code

AI-generated content may be incorrect.

## 2.5 Final Thoughts:

This design and implementation ensures that the Library Management System is scalable, maintainable and user-friendly. It also follows the best practices.

# **Code Explanation:**

## Explanation of key sections of the code:

In this section I will going over the main sections of code which make up the Library Management System. I will focus on the controller, service and repository classes which are the main components of the code. I will also include an image of the project layout showing all the packages and classes which are made. As I will be submitting the code I will focus on the user’s classes and share images of how I setup the controller for the user, service and repository. The loan and book classes are setup very similar so I will not include these in my report but will have showcased them in the screen cast and also of course in my complete code submission.

### Setting up Controllers and Routes

For this assignment I created three controllers one for each entity. Controllers in a RESTful API handle HTTP requests and dispatch them to appropriate service methods, returning responses to the client. Below you can see my UserController and how I configured it:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

* @RestController is the annotation used to mark a class as a controller. With this annotation all methods in the class now return a domain object rather then a view.
* @RequestMapping(“/users”) this annotation defines that all the URLs will be prefixed with “/users”.
* @GetMapping, @PostMapping, @PutMapping and @DeleteMapping are annotations which are used to handle GET, POST, PUT and DELETE requests. Another thing to not is that along with some of these annotations an additional prefix is added to a URL such as /”delete” and then another prefix such as “/{id}” is used. When u see the currley braces this means that the method uses the value which is added to that URL. The @PathVariable allows us to store that in Java using a variable we create in this instance it’s a variable of type Long and the variable is id.

### Service and Repository Layers:

The Service layers of this API handle what’s known as the business logic of the application, while the repository layer interacts with the database. So when a controllers endpoint is called, the method connected to that endpoint calls a method in the service class which then calls the repository layer to make changes to the database or retrieve data from the database.

Here is an example demonstrating these layers:

**Service Layer:**

A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

* @Service marks it as a business logic class.
* @Autowired is used in this case to perform a field injection which means it injects the dependency directly into the class field
* @Transactional this annotation is very useful for updates as it ensures that a user either completely saves or not all. This means that if a run time exception occurs the user will be rolled back and none of the updated fields will be changed. This stops a user from having some of there fields updated but not all of them.

**Repository Layer:**

A screen shot of a computer program

AI-generated content may be incorrect.

* JpaRepository provides CRUD functions and query capabilities.

## Example of a One-to-Many Relationship in the API:

A one-to-many relationship which is implemented in the library management system is between the user and the loan entities. This relationship dictates that a user can have 0 or many loans. Below I will insert images of my User entity and loan entity classes which show how I implemented the one-to-many relationship using spring annotations.

**User Entity:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Loan Entity:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

From the images above you can see I used the @OneToMany and @ManyToOne annotations to establish the relationship in the jpa model. You can also see annotations I used for validation, generating ID values, naming columns and also naming the user entity table “users” as the term user is protect in Java.

## Sample API Responses:



When fetch a list of all users my sample response will give me the users name, email, ID and loans. It will also have the hateoas links at the end. Here is a screenshot of the response given when I make that get request.



This example shows the nested format , clearly reflecting the one-to-many relationship.

# **Challenges and Solutions:**

I faced a few challenges while completing this project. The first challenge was using spring again. I had done one project using spring before for my final year project for my bachelors degree, since then I haven’t had a chance to use spring again. Doing this project I realised I had forgotten a lot of what spring offers and as I worked on this project more and more stuff was coming back to me and also researching differn’t annotations to help me complete steps of this assignment such as the validation annotations made program so much more enjoyable as I now didn’t have to manually do things validation. For all the advantages which spring bring I found it also had its challenges for example:

* **Challenge:** Lombok stopped working in my project half way through for no reason. I had added it as a dependency when I initialised my project using spring initializr. It took me a good hour or more searching the web for answers on why Lombok just stopped working. Eventually I found someone on stack overflow who faced the same problem and was able to tell me what my dependency for Lombok should look like. They also mentioned to change some settings in my compiler.
* **Challenge:** Another challenge was coming up with a project which met the criteria of the brief without picking an idea which was to big and overcomplicated. It took me quite some time to finally come up with a Library management system and work out exactly what classes I needed I order to satisfy the brief.
* **Challenge:** validating data was a big concern for this project as I was worried that if incorrect data was entered into one of the POST or PUT endpoints that the API could crash if the database was to have errors. My solution to this problem was adding built in spring annotations for data validation. This made validating the data simple and time effective.

# **Conclusion:**

Developing the Library Management System API was a significant and enriching experience. It allowed me to apply my theoretical knowledge practically and tackle real-world software development challenges.

## **5.1 Project Outcomes**

Successfully implementing the REST API for managing library operations demonstrated the system's effectiveness in smoothly handling books, users, and transactions. Key features like pagination and advanced error handling enhanced the system's capability to serve a busy library environment efficiently. The use of Data Transfer Objects (DTOs) streamlined responses, improving user interaction with the system.

## **5.2 Learning Experiences**

This project deepened my skills in several areas, including using Spring Boot for API development and JPA for database management. I learned how to design RESTful services incorporating advanced features such as HATEOAS. Addressing challenges such as optimising database access and managing concurrent data operations was crucial in enhancing my understanding of the complexities involved in modern API development.

## **5.3 Final Thoughts**

This project was a profound learning opportunity that emphasised the importance of perseverance and meticulous planning. It significantly improved my problem-solving skills and boosted my confidence in managing complex software development tasks independently. Moving forward, I am eager to apply these skills to new challenges, continuing to learn and grow as a software developer.