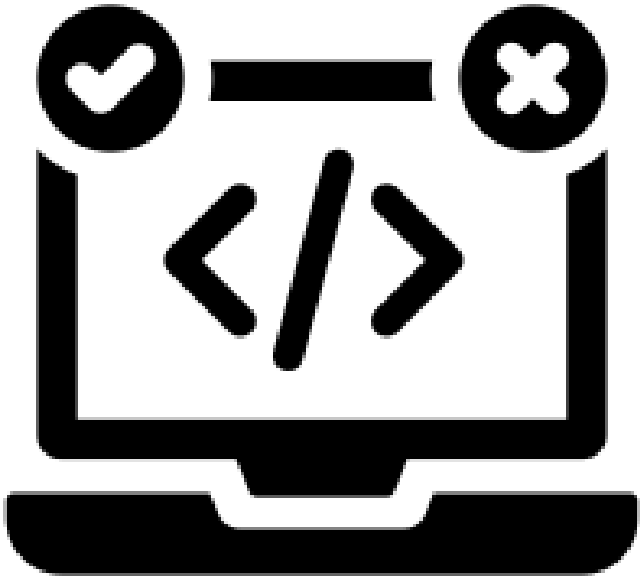
62-51_Integration_DevelopmentLifecycle

TDD Project



Written by: Quentin, Yohann, Lucas

Date: 20/12/24

Table of Contents

Overview

This project implements a library management system divided into two main modules: the Application Module (src folder) and the Test Module (test folder). Each module serves distinct responsibilities to ensure the functionality, scalability, and testability of the application. Below is a detailed explanation of each module and its components.

Application Module (src)

The src folder contains the core application, which is divided into two main layers: the DAL and the API.

The Data Access Layer (DAL) is responsible for managing interactions with the database. It consists of three entities: User, Book, and LibraryTransaction. The API Layer exposes the application's functionalities through HTTP endpoints and handles client-server communication. The API includes a controller (LibraryController.cs) that manages HTTP requests related to library operations, such as book checkouts, returns, and renewals. All business logic is encapsulated within the service layer (LibraryService.cs), ensuring a clean separation of concerns between the API and database interactions.

Test Module (test)

The test folder contains unit tests to verify the correctness and reliability of the application. It is divided into two submodules: controller tests and service tests. The controller tests focus on validating the behavior of controllers in handling HTTP requests and responses, ensuring that they return the correct HTTP status codes. The service tests are designed to validate the business logic implemented in the service layer, ensuring proper functionality and handling of edge cases.

There are no unit tests for the DAL because it solely handles data persistence and retrieval through Entity Framework without any custom logic or implementation.

Setup Instructions

You can access the repository for this project at the following link: [LibraryServices GitHub Repository](https://github.com/GomezQuentin/LibraryServices.git).

1. Open your terminal and run the following command to clone the project repository to your local machine: git clone https://github.com/GomezQuentin/LibraryServices.git
2. Navigate to the folder where the repository was cloned, then open the solution LibraryServices.sln using Visual Studio.

API Documentation:

The Library Services API provides endpoints for managing library operations, including:

- Managing outstanding fees.
- Book checkouts, returns, and renewals.
- Retrieving transaction history.

- Processing payments.
- Checking out several books at the same time

Base URL : http://<your-server-url>/Library

When you run the API, the database will be automatically populated with initial data, allowing you to test the functionality that has been implemented.

Type	Id	Name	FirstName	Fees
User	1	Doe	John	0.00
User	2	Smith	Jane	5.00

Type	Id	Title	Fee Price	Borrowing Days	Available
Book	a	Introduction to C#	1.50	14	True
Book	b	ASP.NET Core in Action	2.00	14	True

Endpoints

Endpoint	HTTP Method	Description	Path/Body Parameters	Responses
/fees/{userId}	GET	Retrieve outstanding fees for a user	**Path Parameters:** userId (integer): ID of the user	**200 OK:** { "UserId": 1, "OutstandingFees": 0 } **404 Not Found:** { "Message": "User not found." } **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}
/checkoutBook	POST	Check out a single book	**Body Parameters:** userId (integer): User ID bookId (string): Book ID	**200 OK:** { "Message": "Book checked out successfully." } **404 Not Found:** { "Message": "Invalid userId or bookId " } **400 Bad Request:** { "Message": "Book not available" } **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}
/return	POST	Return a borrowed book	**Body Parameters:** userId (integer): User ID bookId (string): Book ID	**200 OK:** { "Message": "Book returned successfully." } **404 Not Found:** { "Message": "Invalid userId or bookId " }

				400 Bad Request: { "Message": "No valid checkout record found." } **400 Bad Request:** { "The book cannot be returned because it was not checked out by the current user."} **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}
/renew	POST	Renew a borrowed book	**Body Parameters:** userId (integer): User ID bookId (string): Book ID	**200 OK:** { "Message": "Book renewed successfully. New due date: 20.12.2024" } **404 Not Found:** { "Message": "Invalid userId or bookId " } **400 Bad Request:** { "Message": "The book cannot be returned because it was not checked out by the current user." } **400 Bad Request:** { "Message": "No valid checkout record found for this book and user." } **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}
/checkoutBooks	POST	Check out multiple books	**Body Parameters:** userId (integer): User ID bookIds (List<string>): List of Book IDs	**200 OK:** { "UserId": 1, "Results": { "123": "Checkout successful.", "456": "Book not available." } } **404 Not Found:** { "Message": "User not found." } **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}
/transactions/{userId}	GET	Retrieve transaction history for a user	**Path Parameters:** userId (integer): ID of the user	**200 OK:** { "UserId": 1, "Transactions": { "userId": 1, "transactions": [{ "id": 3, "bookId": "a", "transactionType": "Return", "date": "2024-12-20T07:42:58.6430137" } { "id": 1, "bookId": "a", "transactionType": "Checkout", "date": "2024-12-20T07:42:47.5701094" }]}}

				404 Not Found: { "Message": "User not found." } **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}
/payment	POST	Process payment for outstanding fees	**Body Parameters:** userId (integer): User ID paymentAmount (decimal): Amount paid	**200 OK:** { "Message": "Payment processed successfully. Outstanding fees cleared." } **404 Not Found:** { "Message": "User not found." } **400 Bad Request:** { "Message": "Payment failed: Payment amount does not match the outstanding fees. Outstanding Fees: {user.Fees}" } **500 Internal Server Error:** { "Message": "An internal error occurred. Please try again later."}

Unit Test:

To better understand how to create the unit tests, we began by identifying the functions required for this project and then considered how to test them, including all potential edge cases. Ultimately, we decided to implement the functions mentioned in the API documentation. As described above, we have two test files: one for testing the controllers (HTTP requests) and another for service tests, which focus on core functionalities. The service tests are particularly crucial for ensuring the correct behavior of the application's functionality.

Once the functionalities were planned, we followed the Test-Driven Development (TDD) approach by first implementing the unit tests (Red Phase) and then writing the code to make them pass (Green Phase). After the tests passed, we refactored the code to improve its clarity and efficiency while ensuring it still met all requirements.

LibraryControllerTests

This file contains unit tests for the LibraryController class. The purpose of these tests is to ensure that the API endpoints behave correctly in various scenarios, including normal cases, edge cases, and error handling. The tests utilize the Moq framework to mock the ILibraryService dependency and simulate expected behaviors.

The LibraryControllerTests class initializes a mock ILibraryService and a LibraryController instance. Each test method sets up specific conditions to simulate different scenarios. Tests are grouped based on the functionality of each API endpoint in the LibraryController

Name	Purposes	Tests
GetOutstandingFees	Validates the behavior of the /fees/{userId} endpoint.	User Exists: Ensures 200 OK is returned with the correct fee. User Exists : Return correct fee. User Does Not Exist: Ensures 404 Not Found is returned. Service Exception: Ensures 500 Internal Server Error is returned.
CheckOutBook	Validates the behavior of the /checkoutBook endpoint.	Valid Inputs: Ensures 200 OK is returned for successful checkout. User Not Found: Ensures 404 Not Found is returned. Book Not Found: Ensures 404 Not Found is returned. Book Not Available: Ensures 400 Bad Request is returned. Service Exception: Ensures 500 Internal Server Error is returned.
ReturnBook	Validates the behavior of the /return endpoint	Valid Inputs: Ensures 200 OK is returned for successful return. User Not Found: Ensures 404 Not Found is returned. Book Not Found: Ensures 404 Not Found is returned. No Valid Checkout Record: Ensures 400 Bad Request is returned. Service Exception: Ensures 500 Internal Server Error is returned.
GetUserLibraryTransactions	Validates the behavior of the /transactions/{userId} endpoint.	User Has Transactions: Ensures 200 OK is returned with correct data. User Has Transactions: Return the correct data User Has No Transactions: Ensures 200 OK. User Has No Transactions: Returns an empty List. User Does Not Exist: Ensures 404 Not Found is returned. Service Exception: Ensures 500 Internal Server Error is returned.
ProcessFeePayment	Validates the behavior of the /payment endpoint.	Valid Payment: Ensures 200 OK is returned. User Does Not Exist: Ensures 404 Not Found is returned. Invalid Payment Amount: Ensures 400 Bad Request is returned. Service Exception: Ensures 500 Internal Server Error is returned.
RenewBook	Validates the behavior of the /renew endpoint.	Valid Renewal: Ensures 200 OK is returned with the new due date. User Not Found: Ensures 404 Not Found is returned. Book Not Found: Ensures 404 Not Found is returned. No Valid Checkout Record: Ensures 400 Bad Request is returned. Already Renewed: Ensures 400 Bad Request is returned. Service Exception: Ensures 500 Internal Server Error is returned.
CheckOutBooks	Validates the behavior of the /checkoutBooks endpoint.	All Books Available: Ensures 200 OK Some Books Unavailable: Ensures 200 OK Book Does Not Exist: Ensures 200 OK User Does Not Exist: Ensures 404 Not Found is returned. Empty Book List: Ensures 200 OK Empty Book List: Returns an empty list

		Service Exception: Ensures 500 Internal Server Error is returned.

The test coverage of ProjectContext is at 100%, demonstrating that all implemented functionality has been thoroughly tested.

LibrarySercice Tests

The LibraryServiceTests file ensures the correctness, reliability, and completeness of the business logic in the LibraryService class. It ensures that the service functions operate correctly, handle various edge cases, and adhere to expected behaviors. Each test method targets a specific functionality, emphasizing robustness and correctness.

An in-memory database is used (ProjectContext with UseInMemoryDatabase) to simulate data operations.

Function tested	Test Method Name	Purpose
GetOutstandingFees	GetOutstandingFees_UserExistsWithFees_ReturnsCorrectFees	Ensures the correct fee calculation for a user with outstanding fees
GetOutstandingFees	GetOutstandingFees_UserExistsWithoutFees_ReturnsZero	Verifies that users with no fees return zero as outstanding fees.
GetOutstandingFees	GetOutstandingFees_UserDoesNotExist_ThrowsArgumentException	Validates that non-existent users throw an exception.
GetOutstandingFees	GetOutstandingFees_MultipleUsers_ReturnsCorrectFeesForSpecifiedUser	Ensures fees are correctly calculated for specific users in a multi-user setup.
GetOutstandingFees	GetOutstandingFees_UserHasNegativeFees_ReturnsNegativeValue	Verifies that negative fees are handled correctly.
CheckOutBook	CheckOutBook_BookDoesNotExist_ReturnsError	Verifies error handling when attempting to check out a non-existent book.
CheckOutBook	CheckOutBook_UserDoesNotExist_ReturnsError	Ensures error handling for a non-existent user attempting a checkout
CheckOutBook	CheckOutBook_BookNotAvailable_ReturnsError	Validates error response when attempting to check out an unavailable book.
CheckOutBook	CheckOutBook_ValidInputs_UpdatesAvailability	Ensures book availability updates correctly after a successful checkout
CheckOutBook	CheckOutBook_MultipleBooks_UpdatesAvailability	Verifies correct handling of multiple book checkouts in a single transaction.
ReturnBook	ReturnBook_UserDoesNotExist_ThrowsArgumentException	Ensures an exception is thrown for a return attempt by a non-existent user
ReturnBook	ReturnBook_BookDoesNotExist_ThrowsArgumentException	Verifies error handling for return attempts of non-existent books.
ReturnBook	ReturnBook_ReturnOnTime_NoFeesApplied	Validates no fees are applied when books are returned on time.

ReturnBook	ReturnBook_ReturnLate_FeesApplied	Ensures late fees are correctly applied when books are returned past the due date.
ReturnBook	ReturnBook_BookAlreadyAvailable_ThrowsInvalidOperationException	Validates that a book already marked as available cannot be returned again.
ReturnBook	ReturnBook_NoValidCheckoutRecord_ThrowsInvalidOperationException	Ensures returns without a valid checkout record fail with the correct error message.
ReturnBook	ReturnBook_UserRenewsAndReturnsOnTime_NoLateFees	Ensures renewals extend the borrowing period without fees if returned on time.
ReturnBook	ReturnBook_UserRenewsAndReturnsLate_AccruesLateFees	Validates late fees are applied correctly even after a renewal.
GetUserLibraryTransactions	GetUserLibraryTransactions_UserHasTransactions_ReturnsTransactionList	Ensures correct retrieval of library transactions for a user with transactions.
GetUserLibraryTransactions	GetUserLibraryTransactions_UserHasNoTransactions_ReturnsEmptyList	Verifies users with no transactions return an empty list.
GetUserLibraryTransactions	GetUserLibraryTransactions_UserDoesNotExist_ThrowsArgumentException	Ensures an exception is thrown for non-existent users.
ProcessFeePayment	ProcessFeePayment_ValidPayment_ClearsFees	Ensures fees are cleared after a valid payment.
ProcessFeePayment	ProcessFeePayment_UserDoesNotExist_ThrowsArgumentException	Verifies error handling for non-existent users attempting a fee payment.
ProcessFeePayment	ProcessFeePayment_InvalidPaymentAmount_ReturnsErrorMessage	Validates that incorrect payment amounts return an appropriate error message.
RenewBook	RenewBook_ValidRenewal_RenewsSuccessfully	Ensures successful renewal of a book.
RenewBook	RenewBook_UserDoesNotExist_ThrowsArgumentException	Verifies error handling for renewal attempts by a non-existent user.
RenewBook	RenewBook_BookDoesNotExist_ThrowsArgumentException	Validates error handling for renewal attempts of non-existent books.
RenewBook	RenewBook_NoCheckoutRecord_ThrowsInvalidOperationException	Ensures renewals without a valid checkout record fail with the correct error message.
RenewBook	RenewBook_AlreadyRenewed_ReturnsRenewalFailedMessage	Verifies that books already renewed cannot be renewed again.
CheckOutBooks	CheckOutBooks_AllBooksAvailable_ReturnsSuccessForAll	Ensures all books are checked out successfully when available.

CheckOutBooks	CheckOutBooks_SomeBooksUnavailable_ ReturnsPartialSuccess	Verifies that unavailable books result in partial success for checkouts.
CheckOutBooks	CheckOutBooks_BookDoesNotExist_Returns ErrorForMissingBooks	Ensures errors are returned for non-existent books during checkouts.
CheckOutBooks	CheckOutBooks_UserDoesNotExist_Throws ArgumentException	Validates error handling for non-existent users attempting multiple book checkouts.
CheckOutBooks	CheckOutBooks_EmptyBookList_ReturnsE mptyResults	Verifies that an empty book list results in no checkouts.
CheckoutBooks	CheckOutBooks_BookUnavailable_Handles InvalidOperationException	Ensures appropriate error handling for unavailable books during multiple checkouts.

The test coverage of LibraryService.cs is at 99.76%. This high level of coverage ensures that the implemented code has been rigorously tested and is functioning as intended, covering both standard use cases and edge cases effectively.

Ressources

- Microsoft. (n.d.). *UseInMemoryDatabase method*. Microsoft Learn. Retrieved December 20, 2024, from https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.inmemorydbcontextoptionsextensions.useinmemorydatabase?view=efcore-9.0
- Extinctsion. (n.d.). *Comprehensive testing in .NET 8 using Moq and in-memory databases*. DEV Community. Retrieved December 20, 2024, from https://dev.to/extinctsion/comprehensive-testing-in-net-8-using-moq-and-in-memory-databases-ioo
- Widmer, A. (n.d.). *HES-SO-VS_62-51_Integration_DevelopmentLifecycle*. Lecture material on test-driven development (TDD). HES-SO Valais-Wallis.