

# Module 1

## What to learn

- Introduction
- Understanding Visual Studio IDE
- Creating Hello World Application
- Debugging
- Datatype
  - Value Type
  - Reference Type
- Variables
- Arrays
- If statement/Switch
- Operators
- Looping Constructs
- Constants (enum)

## Practice Exercise

### Practice 1

Do the hands on the things provided in videos and provided in the following url  
<https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console?view=vs-2019>

### Practice 2

## Introduction to C# and Visual Studio IDE

### Explore Visual Studio:

Open Visual Studio, create a new C# Console Application, and navigate the Solution Explorer, Properties, and Output Window.

### Hello World Application:

Create a Console Application that prints "Hello, World!" to the console.

### Modify Hello World:

Update the program to ask the user for their name and greet them with "Hello, [Name]!".

### Using Comments:

Add single-line and multi-line comments to the Hello World program explaining each part of the code.

### Building and Running:

Build and run the Hello World application, observing the build output and errors if any.

### Practice 3

## Debugging

### **Adding Breakpoints:**

Add a breakpoint to the Hello World application. Debug it and observe the flow of execution.

### **Step Over/Into:**

Write a program with a simple function and use "Step Into" and "Step Over" in the debugger.

### **Variable Watch:**

Declare and initialize a few variables. Use the debugger to watch their values during execution.

### **Error Simulation:**

Introduce a deliberate error (like a division by zero). Debug the program to understand the error.

### **Debugging Logic Errors:**

Write a program to calculate the sum of two numbers. Intentionally make a logic error and debug it to fix the issue.

### **Practice 4**

## **Data Types**

### **Value Types:**

Write a program declaring int, float, and char variables. Assign and print their values.

### **Reference Types:**

Create a program using a string and an array. Assign values and print them.

### **Type Conversion:**

Perform explicit and implicit type conversions between int and double.

### **Nullable Types:**

Create a nullable int variable, assign values, and handle cases where it is null.

### **Dynamic Type:**

Write a program to declare a dynamic variable, assign different types of values, and print them.

### **Practice 5**

## **Variables**

### **Declaring Variables:**

Declare and initialize variables of different types, and print them using string interpolation.

### **Arithmetic with Variables:**

Write a program to perform arithmetic operations using two variables.

### **Constant Variables:**

Declare a const variable for the value of Pi and use it to calculate the circumference of a circle.

### **Scope of Variables:**

Write a program demonstrating local and global variable scopes.

### **Swapping Variables:**

Write a program to swap the values of two variables without using a third variable.

### **Practice 6**

## **Arrays**

### Single-Dimensional Array:

Write a program to store 5 student marks in an array and display them.

### Multi-Dimensional Array:

Create a 2D array of integers and initialize it with values. Print its elements using nested loops.

### Jagged Array:

Create a jagged array to store the names of students in different classes and print them.

### Array Length:

Write a program to find the length of an array entered by the user.

### Array Sorting:

Create an array of integers, sort it in ascending order, and display the sorted values.

## Assignment Exercise

### Assignment 1 (65cdfdab7ba36e06448762c8)

Print sum of all the even numbers

### Assignment 2 (65cdfdab7ba36e06448762c9)

Store your name in one string and find out how many vowel characters are there in your name.

### Assignment 3 (65cdfdab7ba36e06448762ca)

Create a weekday enum and accept week day number and display week day.

### Assignment 4 (65cdfdab7ba36e06448762cb)

Accept 10 student Name,Address,Hindi,English,Maths Marks ,do the total and compute Grade.

Note do it with Array and display the result in grid format

### Assignment 5 (65cdfdab7ba36e06448762cc)

Accept Age from user, if age is more than 18 eligible for vote otherwise it should be displayed as not eligible. Do it with ternary operator

### Assignment 6

## Assignment: Employee Management System

Create a Console Application in C# that serves as a basic Employee Management System. The program should integrate the concepts learned in Introduction to C#, Debugging, Data Types, Variables, Arrays, Conditional Statements, Loops, and Constants/Enums.

### Objective:

Build a program that:

- Allows users to manage employee records (add, view, search, and calculate salaries).
- Includes logic to determine bonuses based on performance ratings.

### Requirements:

#### 1. Introduction to Visual Studio & Debugging:

**Create the project:** Start a new Console Application in Visual Studio named EmployeeManagementSystem.

**Use Debugging tools:** Add breakpoints and debug the application as needed.

#### 2. Employee Class (Data Types, Variables, and Reference Types):

Create a class Employee with the following properties:

int ID (Value Type)  
string Name (Reference Type)  
string Department  
double Salary  
int PerformanceRating (1-5 scale)

### 3. Menu (If Statements/Switch):

Create a menu-driven system with the following options:

Add an Employee.  
Display All Employees.  
Search Employee by ID.  
Calculate Bonus for Employees.  
Exit.

### 4. Business Logic (Operators and Arrays):

**Add Employee:** Store employee details in an array or a list (use an array of size 10 for simplicity).  
**Display All Employees:** Loop through the array and display details of each employee.  
**Search Employee by ID:** Use a loop to find the employee with the matching ID and display their details.  
**Calculate Bonus:** Use this formula:  $\text{Bonus} = \text{Salary} \times (\text{Performance Rating} / 10)$   
Display each employee's bonus.

### 5. Constants/Enums:

Use an **enum** for department names (e.g., HR, IT, Finance, Sales).  
Use a **constant** for the base salary multiplier (e.g., `const double BaseMultiplier = 0.1;`).

### 6. Validation (Conditional Statements):

Validate user input (e.g., Performance Rating should be between 1 and 5).  
Ensure the array doesn't exceed its size (handle cases when trying to add more than 10 employees).

Welcome to Employee Management System

- 
1. Add Employee
  2. Display All Employees
  3. Search Employee by ID
  4. Calculate Bonus for Employees
  5. Exit
- 

Enter your choice: 1

Enter Employee ID: 101

Enter Name: John Doe

Enter Department (HR/IT/Finance/Sales): IT

Enter Salary: 50000

Enter Performance Rating (1-5): 4

Employee Added Successfully!

-----  
Enter your choice: 4

ID: 101, Name: John Doe, Bonus: 2000

## Deliverables:

Submit the C# project folder containing the complete solution.

Include the following:

**Employee.cs:** Class definition for Employee.

**Program.cs:** Main program logic

## Online Reference

<https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console?view=vs-2019>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/>

## .NET Core Web API

WEB API (old)

Authentication And Authorization (WEBAPI)(old)

FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 2

## What to learn

Understanding Classes  
Classes Demo  
Properties  
Constructor  
Static Methods  
Extend a simple C# console app using class library

## Practice Exercise

### Practice 1

Do the hands C# video and practical given on <https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console-part-2?view=vs-2019>

### Practice 2

<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/introduction-to-classes>

### Practice 3

<https://docs.microsoft.com/en-us/dotnet/csharp/properties>

### Practice 4

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-constructors>

### Practice 5

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>

### Practice 6

## Understanding Classes

### Define a Class:

Write a simple class named Person with fields Name and Age.

### Create an Object:

Create an object of the Person class in the Main method and assign values to its fields.

### Instance vs. Class:

What is the difference between an instance of a class and the class itself? Explain with an example.

### Access Modifiers:

Write a program to demonstrate public and private access modifiers in a class.

### Class Usage:

Create a Car class with Brand, Model, and Price as fields. Use this class to print details of a car.

### Practice 7

### Field Initialization:

Write a Product class with fields Name, Category, and Price. Initialize them in the Main method

and print the details.

#### **Encapsulation:**

Modify the Product class to make its fields private and create methods to set and get values.

#### **Array of Objects:**

Create an array of Product objects, initialize 3 products, and print their details.

#### **Methods in Classes:**

Add a method CalculateDiscount to the Product class that calculates a discount based on price.

#### **Class Relationship:**

Create a class Order that contains a list of Product objects. Use the relationship to display details of all products in an order.

### **Practice 8**

## **Properties**

#### **Auto-Implemented Properties:**

Create a class Employee with Name, ID, and Salary as auto-implemented properties.

#### **Custom Properties:**

Add a custom property AnnualSalary in the Employee class that calculates and returns Salary \* 12.

#### **Read-Only Property:**

Add a read-only property Department in the Employee class initialized in the constructor.

#### **Validation with Properties:**

Add validation in the Salary property to ensure it cannot be negative.

#### **Default Value in Properties:**

Use a property in a class that has a default value for Department.

### **Practice 9**

## **Constructor**

#### **Parameterized Constructor:**

Write a class Student with fields Name and Age. Use a parameterized constructor to initialize these fields.

#### **Overloading Constructors:**

Add a second constructor to the Student class that initializes only the Name field, setting a default value for Age.

#### **Default Constructor:**

Create a default constructor in the Student class that initializes default values for all fields.

#### **Constructor Chaining:**

Demonstrate constructor chaining in the Student class by calling one constructor from another.

#### **Real-Life Scenario:**

Write a class Book that requires Title and Author in its constructor and prints these details in a method.

### **Practice 10**

### Utility Class:

Create a static class MathUtilities with a static method Add that takes two numbers and returns their sum.

### Static Counter:

Write a class with a static field to keep track of the number of objects created from the class.

### Calling Static Methods:

Create a static method Greet in a class Helper and call it without creating an object of the class.

### Non-Static with Static:

Create a non-static method that calls a static method from within the same class.

### Static vs. Instance Methods:

Demonstrate the difference between static and instance methods using a simple class.

## Practice 11

# Mini-Project: Extend a Console App with a Class Library

### Scenario:

You are tasked with building a **Student Management System** that uses a **class library** to handle business logic.

### Steps:

#### Create a Class Library:

- Create a new project for a Class Library named BusinessLogic.

- Add a class StudentManager with the following methods:

  - AddStudent: Accepts Name and Age and adds a student to a list.

  - GetStudentDetails: Returns details of all students.

  - CalculateAverageAge: Calculates the average age of students.

#### Main Console Application:

- Create a Console App that references the BusinessLogic class library.

- In the Main method, use StudentManager to:

  - Add at least three students.

  - Display all student details.

  - Calculate and display the average age.

### Student Management System

-----  
1. Add Student

2. View All Students

3. Calculate Average Age

4. Exit  
-----

Enter your choice: 1

Enter Name: John

Enter Age: 20

Enter your choice: 2

ID: 1, Name: John, Age: 20



Enter your choice: 3

Average Age: 20

## Assignment Exercise

### Assignment 1 (65cdfdab7ba36e06448762ce)

Create a reference type called Person. Populate the Person class with the following properties to store the following information: - First name - Last name - Email address - Date of birth Add constructors that accept the following parameter lists: - All four parameters - First, Last, Email - First, Last, Date of birth Add read-only properties that return the following computed information: - Adult - whether or not the person is over 18 - Sun sign - the traditional western sun sign of this person - Chinese sign - the chinese astrological sign (animal) of this person - Birthday - whether or not today is the person's birthday - Screen name - a default screen name that you might see being offered to a first time user of AOL or Yahoo (e.g. John Doe born on February 25th, 1980 might be jdoo225 or johndoe022580) Access these things from Console Application in the Main Function. Accept this data for 5 person and display the same. Means create an object Array of 5 size and accept these details and display these details in tabular format.

### Assignment 2

## Assignment: Employee Payroll Management System

### Objective:

Build a C# Console Application for an Employee Payroll Management System. Use a Class Library for the business logic. The system will allow users to manage employees, calculate payroll details, and demonstrate key concepts such as classes, properties, constructors, and static methods.

### Requirements:

#### 1. Create a Class Library (BusinessLogic):

Create a Class Library project named PayrollLibrary with the following components:

##### Class: Employee

**Fields:** EmployeeID, Name, Department, BasicSalary, HRA, DA, Bonus.

##### **Properties:**

Auto-implemented properties for EmployeeID, Name, and Department.

Validated properties for BasicSalary, HRA, DA, and Bonus to ensure they are non-negative.

##### **Constructor:**

Parameterized constructor to initialize EmployeeID, Name, and Department.

Default constructor to set default values.

##### Class: PayrollManager

**Static Method:** CalculateNetSalary(Employee employee)

Business Logic:  $\text{NetSalary} = \text{BasicSalary} + \text{HRA} + \text{DA} + \text{Bonus}$   
 $\text{NetSalary} = \text{BasicSalary} + \text{HRA} + \text{DA} + \text{Bonus}$

**Instance Method:** AddEmployee(Employee employee)

Adds an employee to a list.

**Instance Method:** DisplayEmployees()

Displays all employee details.

## Main Console Application:

Create a **Console Application** project named EmployeePayrollApp. Add a reference to the PayrollLibrary class library.

**Features in the Console App:**

### Main Menu

1. Add Employee
2. View All Employees
3. Calculate and Display Net Salary
4. Exit

**Add Employee:**

Accept EmployeeID, Name, Department, and BasicSalary from the user.

Automatically calculate HRA (20% of Basic Salary), DA (10% of Basic Salary), and Bonus (5% of Basic Salary).

Use the PayrollManager.AddEmployee method to add the employee to the system.

**View All Employees:**

Display a list of all employees using the PayrollManager.DisplayEmployees method.

**Calculate and Display Net Salary:**

Select an employee by ID.

Use the PayrollManager.CalculateNetSalary method to compute and display the net salary.

## Deliverables:

A complete **C#** solution with:

PayrollLibrary Class Library.

EmployeePayrollApp Console Application.

Code should include:

Proper use of **classes, properties, constructors, and static methods**.

Business logic implemented in the class library for modularity and reusability.

## Expected Learning Outcomes:

Develop an understanding of integrating class libraries in a console application.

Apply object-oriented programming principles to solve intermediate business problems.

Practice modular design and reusable components.

### Online Reference

<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/introduction-to-class>

<https://docs.microsoft.com/en-us/dotnet/csharp/properties>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/using-static-members>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 3

## What to learn

Object Oriented Programming  
Inheritance  
Polymorphism  
Encapsulation  
Abstraction

## Practice Exercise

### Practice 1

Do the hands on the things provided in video and url <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/object-oriented-programming>

### Practice 2

## Inheritance

### Single Inheritance:

Create a base class Vehicle with properties Make and Model. Derive a class Car that adds a property FuelType. Write a program to initialize a Car object and display all its details.

### Method Overriding:

In the Vehicle class, add a method DisplayInfo() that prints the Make and Model. Override this method in the Car class to include FuelType. Call both versions using a Car object.

### Constructor Chaining:

Demonstrate constructor chaining between the Vehicle and Car classes. Pass values for Make, Model, and FuelType while creating a Car object.

### Multilevel Inheritance:

Add a new class ElectricCar derived from Car, with a property BatteryCapacity. Demonstrate the initialization and display of all details across three levels of inheritance.

### Real-World Scenario:

Create a Person class with Name and Age. Derive a Student class that adds StudentID and Course. Derive a GraduateStudent class that includes GraduationYear. Implement a program to manage graduate student details.

### Practice 3

## Polymorphism

### Compile-Time Polymorphism (Overloading):

Create a class MathOperations with overloaded methods Add that:

Accepts two integers.  
Accepts three integers.

Accepts two doubles.

Test all overloads in a program.

### **Run-Time Polymorphism (Virtual Methods):**

Create a base class Shape with a virtual method CalculateArea(). Derive Circle and Rectangle classes that override CalculateArea(). Use polymorphism to calculate areas for different shapes.

### **Abstract Class with Polymorphism:**

Create an abstract class Employee with an abstract method CalculateSalary(). Derive classes FullTimeEmployee and PartTimeEmployee that implement this method. Use polymorphism to calculate salaries for both types.

### **Interface-Based Polymorphism:**

Define an interface IDiscount with a method ApplyDiscount(). Implement this interface in RegularCustomer and PremiumCustomer classes with different discount rates. Demonstrate polymorphism by applying discounts to both customer types.

### **Real-World Scenario:**

Create a Payment class with a virtual method ProcessPayment(). Derive CreditCardPayment and UPIPayment classes that override this method. Use a List<Payment> to process multiple types of payments in a single program.

### **Practice 4**

## **Encapsulation**

### **Encapsulation with Properties:**

Write a class BankAccount with private fields AccountNumber, Balance, and AccountHolder. Use properties to encapsulate these fields and add validation for setting Balance.

### **Encapsulation with Business Logic:**

Add a method Deposit and Withdraw to the BankAccount class. Ensure the Withdraw method prevents overdrafts and updates the balance.

### **Read-Only Properties:**

Create a Product class with a read-only property DiscountedPrice that is calculated as Price - Discount. Ensure Discount is validated using encapsulation.

### **Encapsulation with Method Calls:**

Write a class Order that contains private fields for OrderID, Items, and TotalAmount. Use encapsulation to manage adding items and updating the total amount.

### **Real-World Scenario:**

Create a Student class with private fields for StudentID, Name, Marks. Use methods to add marks for subjects and calculate the percentage. Ensure marks cannot exceed a specified range.

### **Practice 5**

## **Abstraction**

### **Abstract Class:**

Create an abstract class Appliance with abstract methods TurnOn() and TurnOff(). Derive Fan and WashingMachine classes that implement these methods.

### **Interface Implementation:**

Define an interface IReport with a method GenerateReport(). Implement it in PDFReport and

ExcelReport classes. Demonstrate abstraction by using only the interface reference to generate reports.

#### **Abstract Class with Business Logic:**

Write an abstract class Shape with a method CalculatePerimeter() and an abstract method CalculateArea(). Implement it in Square and Triangle classes. Demonstrate abstraction by calculating areas and perimeters for both shapes.

#### **Real-World Scenario with Abstraction:**

Create an abstract class Loan with properties LoanAmount and InterestRate. Add an abstract method CalculateEMI(). Implement it in HomeLoan and CarLoan classes, each with their specific EMI formula.

#### **Combining Interface and Abstract Class:**

Create an abstract class Vehicle with a property Speed and an abstract method DisplayDetails(). Create an interface IFuelEfficiency with a method CalculateEfficiency(). Implement both in a HybridCar class.

### **Assignment Exercise**

#### **Assignment 1**

Implement all the oops concept for Employee payroll system.

#### **Assignment 2**

Create above assignment's class hierarchy, implement inheritance and polymorphism.

#### **Assignment 3**

### **Assignment: Inventory Management System**

#### **Objective:**

Build a **C# Console Application** for an **Inventory Management System** that demonstrates **Encapsulation, Abstraction, Inheritance, and Polymorphism**. The system should manage different types of products, calculate inventory value, and handle business-specific logic like discounts and tax calculations.

### **Requirements:**

#### **1. Encapsulation**

Encapsulate fields such as ProductID, Name, Price, and Stock within a base class Product using properties.

Add validation for Price and Stock to ensure they are non-negative.

#### **2. Abstraction**

Create an **abstract class** Product with:

Abstract method CalculateTax() to calculate tax based on product type.

Abstract method ApplyDiscount() for different discount rules.

Concrete method CalculateTotalValue() to calculate the total value of stock (Price \* Stock).

#### **3. Inheritance**

Derive the following classes from Product:

Electronics

Specific tax calculation: Tax = 18% of Price.

Discount: 10% discount on products priced above 5000.

#### Groceries

Specific tax calculation: Tax = 5% of Price.

Discount: No discount for groceries.

#### Clothing

Specific tax calculation: Tax = 12% of Price.

Discount: 15% discount on products priced above 2000.

## 4. Polymorphism

Use polymorphism to calculate and display tax, discount, and inventory value for different product types using a list of Product references.

#### Online Reference

<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/>

[https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/abs](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/abstract-classes-and-abstract-interfaces)

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/poly>

## .NET Core Web API

### WEB API (old)

### Authentication And Authorization (WEBAPI)(old)

### FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 4

## What to learn

Collection

Array

Stack Queue

Generic Collection

List

Stack

Queue

Dictionary (map key value pair)

Using Generic collection Interface (ICollection Interface)

Operation on collection sort, search, Index, Add, count remove.

## Practice Exercise

### Practice 1

Store a product information in map object. Key will be a product item and value will be the price of that product. Search the product by product name.

### Practice 2

## Collections

Write a program to find duplicate elements in a collection and return them along with their frequency.

Implement a method to merge two collections of different types into a single collection of key-value pairs.

Create a collection of objects representing employees and group them by their department using LINQ.

Implement a solution to filter out items from a collection based on multiple conditions (e.g., price > 100 and category = "Electronics").

Write a function to flatten a collection of nested collections into a single collection.

### Practice 3

## Array

Implement a method to find the subarray with the maximum sum from a given array.



Write a program to rotate an array by  $k$  positions to the right without using extra space.

Create a solution to find all unique triplets in an array that sum to a specific value.

Develop a function that takes an array of integers and reorders it such that even numbers come before odd numbers.

Write a method to determine if two arrays are permutations of each other.

#### Practice 4

### Stack and Queue

Using a stack, write a function to check if a given string of brackets is balanced.

Design a queue implementation where the maximum element can be retrieved in  $O(1)$  time.

Simulate a browser's forward and backward navigation functionality using two stacks.

Write a program to reverse the order of elements in a queue without using additional queues.

Implement a priority queue to process customer orders based on urgency levels.

#### Practice 5

### Generic Collections

Implement a generic method to merge two collections into one while removing duplicates.

Create a generic method to find the median of a collection of numeric types.

Write a program to implement a custom generic collection that supports filtering, sorting, and mapping.

Design a generic collection to handle caching with time-to-live (TTL) for each entry.

Implement a generic method to shuffle the elements of a collection randomly.

#### Practice 6

### List

Write a method to find and remove all duplicate elements from a list without using LINQ.

Develop a function to split a list into sublists of a given size.

Create a program to merge two sorted lists into one sorted list.

Write a method to shift elements of a list cyclically to the left by a given number of positions.

Implement a function to find the intersection of two lists without using built-in methods.

#### Practice 7

### Stack

Write a program to sort a stack using only another stack and no additional data structures.

Create a method to evaluate a postfix mathematical expression using a stack.

Design a stack that supports retrieving the minimum element in  $O(1)$  time.

Implement a function to reverse a string using a stack.

Write a stack-based solution to convert an infix expression to a postfix expression.

#### Practice 8

### Queue

Implement a circular queue that supports dynamic resizing.

Create a method to merge two sorted queues into a single sorted queue.

Write a program to simulate a ticket booking system using a queue, prioritizing VIP customers.

Develop a solution to reverse the first  $k$  elements of a queue without altering the rest.

Design a queue-based implementation for task scheduling with priorities.

#### Practice 9

### Dictionary (Map Key-Value Pair)

Implement a method to find the most frequently occurring value in a dictionary.

Write a function to merge two dictionaries, combining values of common keys into a list.

Create a program to group words by their lengths using a dictionary.

Design a dictionary to store student grades and calculate the average grade for each student.

Write a method to find all keys in a dictionary whose values satisfy a specific condition.

#### Practice 10

### Using Generic Collection Interfaces (ICollection)

Implement a method using `ICollection` to remove items from a collection based on a predicate.

Write a function that accepts an `ICollection` and returns the second largest element.

Create a generic method using ICollection to compute the union of two collections.

Develop a method to clone an ICollection into a new collection of the same type.

Write a function using ICollection to implement pagination for large data sets.

## Practice 11

# Operations on Collection (Sort, Search, Index, Add, Count, Remove)

Write a method to sort a collection of employees by multiple fields (e.g., age and then name).

Create a function to search for an element in a sorted collection using binary search.

Implement a method to insert an item at a specific index in a collection and maintain sorted order.

Write a solution to count the occurrences of each unique element in a collection.

Develop a method to remove items from a collection that are not present in another collection.

## Assignment Exercise

### Assignment 1

Rambo Rental Bikes is looking for developing a system to calculate the rentals of the bikes. System should accept the customer details, bike details and calculate the rental charges. DESCRIPTION OF PROJECTS System allows users to add customer details with bike rented. It computes rent for each customer. Systems displays the Bike details with summation of days of hire and rental payment. FUNCTIONALITY AND TASK Define a class called Mobike with the following description: Instance variables/data members: BikeNumber – to store the bike's number PhoneNumber – to store the phone number of the customer Name – to store the name of the customer Days – to store the number of days the bike is taken on rent o charge – to calculate and store the rental charge Member methods: void Input( ) – to input and store the detail of the customer. void Compute( ) – to compute the rental charge void display( ) – to display the details in the following format: Bike No. PhoneNo No. of days Charge The rent for a mobike is charged on the following basis: First five days Rs 500 per day Next five days Rs 400 per day Rest of the days Rs 200 per day Use collection Framework to store 10 Customer Details. Implement List operation add, delete, edit and search functionality

### Assignment 2

# Assignment: Comprehensive Data Structures and Collections in C#

## Objective:

The goal of this assignment is to apply and integrate concepts from **Collections**, **Arrays**, **Stacks**, **Queues**, **Generic Collections**, and **Dictionaries** to solve real-world problems. Each task focuses on specific functionality and should be completed in 3–4 hours.

## Instructions:

- Use **C#** for implementing the solutions.
- Each question represents a task that builds towards a unified application.
- Submit your code along with brief explanations of your logic for each task.
- Make sure your code is modular and follows best practices.

## Scenario:

You are developing a **library management system** that involves various data structures to manage books, users, and borrowing records. Implement the following tasks as part of the system:

## Tasks

### Task 1: Manage Book Collection (Collections)

Create a collection to store book details (Book ID, Title, Author, and Availability).

Implement methods to:

- Add a new book.
- Find books by a specific author.
- Remove a book by its ID.

### Task 2: Efficient Search in Book List (Array)

Use an array to store book IDs (integer values).

Implement a function to:

- Find the index of a given Book ID using **binary search**.
- Check if the array is sorted; if not, sort it before performing the binary search.

### Task 3: Book Borrowing (Stack and Queue)

**Stack:** Implement a "Recent Borrowed Books" feature to track the last 5 books borrowed by any user. Use a stack to store and display them in reverse order of borrowing.

**Queue:** Create a waiting queue for books that are currently unavailable. Implement methods to:

Add a user to the waiting queue.

Serve the next user in line when the book becomes available.

#### Task 4: Generic Collection for Borrowing Records (Generic Collections)

Implement a **generic class** `BorrowingRecord<T>` to store borrowing details for any entity type (e.g., books, journals).

Add methods to:

Record a new borrowing transaction.

Retrieve all borrowings for a specific user.

#### Task 5: User Management (List)

Use a list to manage user details (User ID, Name, and Borrow Count).

Implement methods to:

Find the user with the maximum number of borrowings.

Remove users with zero borrowings from the list.

#### Task 6: Recently Returned Books (Stack)

Create a stack to store the details of books returned recently.

Implement a method to transfer all books from this stack to the "Available Books" collection when the stack reaches 10 items.

#### Task 7: Track User Reservations (Queue)

Implement a queue-based system to manage user reservations for books.

Provide methods to:

Add a reservation for a user.

Remove a reservation once it has been fulfilled.

#### Task 8: Book Search by Keywords (Dictionary)

Use a dictionary where the key is a keyword (e.g., "science", "history"), and the value is a list of book titles.

Implement methods to:

Add a new keyword and its associated books.

Search for all books under a specific keyword.

#### Task 9: Collection Operations (Sort, Search, Remove)

Implement methods to:

Sort the book collection alphabetically by title.

Search for a book by title using a linear search.

Remove books that haven't been borrowed in the last year.

#### Task 10: Integrating Everything (Final Application)

Library Management System

-----

1. Add a Book
2. Search Book by Author
3. Borrow a Book
4. Return a Book
5. View Borrowing Records
6. Manage User Accounts
7. View Waiting Queue
8. Search Books by Keyword
9. Sort Book Collection
10. Exit

Online Reference

No online Reference

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 5

## What to learn

Event  
Delegate  
Func Delegate  
Lambda

## Practice Exercise

### Practice 1

do the hands on video provides

### Practice 2

## Event

Implement a custom event BookAdded in a library system that triggers whenever a new book is added. Display the book details when the event is triggered.

Create a program where an event is raised when a stock price crosses a predefined threshold. Notify all subscribers of the change.

Develop a Timer class that raises an event every second. Create a subscriber class that listens to this event and displays the current time.

Write a program where a BankAccount class raises an InsufficientFunds event when a withdrawal exceeds the balance. Implement multiple subscribers to log the event and alert the user.

Implement an event system for a Door class where an event DoorOpened is raised when the door state changes. Have different subscribers respond to the event (e.g., log the state, notify security).

### Practice 3

## Delegate

Write a program using a delegate to sort a list of integers in both ascending and descending order. The order should be decided dynamically based on the delegate passed.

Create a Calculator class that performs addition, subtraction, multiplication, and division using delegates. Allow the user to select the operation at runtime.

Implement a delegate in a FileProcessor class to allow dynamic filtering of file names based on file extensions.

Create a NotificationSystem class that sends different types of notifications (email, SMS, push notification) using a delegate.  
Write a program where a delegate is used to perform different transformations (e.g., capitalize, reverse) on a string.

#### Practice 4

### Func Delegate

Implement a program using Func<T, TResult> to calculate the area of different shapes (circle, rectangle, triangle) based on user input.  
Create a dictionary of operations (add, subtract, multiply, divide) implemented as Func<int, int, int> delegates. Allow the user to choose an operation and provide input.  
Use Func<int, bool> to filter a list of numbers for prime numbers and display the result.  
Write a program using Func<string, string> to transform a list of strings (e.g., convert to uppercase, trim whitespace, append a suffix).  
Implement a method that takes a Func<int, int, bool> to compare two integers and return true if the condition (e.g., greater than, equal) is satisfied.

#### Practice 5

### Lambda

Write a program to find all even numbers in a list using a lambda expression with List<T>.Where().  
Use a lambda expression to sort a list of strings by their lengths.  
Implement a method that takes a list of integers and a lambda function to filter numbers based on a condition (e.g., greater than 10, divisible by 3).  
Create a dictionary of products and prices. Use a lambda expression to find the most expensive product.  
Write a program to group a list of words by their first letter using a lambda expression and GroupBy().

#### Assignment Exercise

##### Assignment 1

Compute area of rectangle using func delegate

##### Assignment 2

Compute add of two number using lambda expression

##### Assignment 3



# Comprehensive Question: Event, Delegate, Func Delegate, and Lambda Integration

## Scenario:

You are building a **task scheduling system** for a productivity application. The system manages tasks, notifies users of task deadlines, and provides customizable filters for viewing tasks.

## Objective:

Implement a **TaskManager** application that integrates **events**, **delegates**, **Func delegates**, and **lambda expressions**.

## Requirements:

### 1. Task Management Using Delegates

Create a **Task** class with the following properties:

- TaskId (int)
- Title (string)
- Deadline (DateTime)
- Priority (enum: Low, Medium, High)
- IsCompleted (bool)

Implement a **TaskManager** class with the following methods:

**AddTask:** Add a task to the list.

**MarkTaskCompleted:** Use a delegate to update the **IsCompleted** property of a task based on its **TaskId**.

**SortTasks:** Sort tasks by a user-specified criterion (e.g., **Deadline**, **Priority**) using a delegate.

### 2. Notification Using Events

Add an event **TaskDeadlineApproaching** in the **TaskManager** class, triggered when a task's deadline is within 24 hours.

Create a **NotificationService** class that subscribes to this event and displays a message with task details.

### 3. Filtering Tasks Using Func Delegates

Implement a method **FilterTasks** in **TaskManager** that takes a **Func<Task, bool>** to filter tasks dynamically. Examples:

- Get all tasks with high priority.

- Get all incomplete tasks.

- Get all tasks due within a week.

### 4. Lambda Expressions for Custom Operations

Use lambda expressions to:

Sort tasks by title length.

Find tasks that contain a specific keyword in the title.

Task Manager System

-----

1. Add a New Task
2. Mark a Task as Completed
3. View All Tasks
4. Filter Tasks by Custom Criteria
5. Trigger Deadline Notifications
6. Exit

Online Reference

No online Reference

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 6

## What to learn

- LINQ
- LINQ Query
- LINQ Methods
- Query Operators
  - Where
  - OfType
  - OrderBy
  - GroupBy, ToLookup
  - Join
  - GroupJoin

## Practice Exercise

### Practice 1

Write a LINQ query to find all even numbers from a list of integers and display them in ascending order.

Create a LINQ query to calculate the average salary of employees whose job title is "Manager".

Using LINQ, retrieve the top 3 highest-scoring students from a list of scores.

Implement a LINQ query to find the distinct departments in a company database.

Use LINQ to create a new list of objects by projecting fields from an existing list (e.g., extract FirstName and LastName into a FullName property).

### Practice 2

Write a LINQ query to retrieve employees whose age is between 25 and 35, sorted by their names.

Using LINQ, find all products that are out of stock from a product list.

Implement a query to group a collection of words by their first letter and count how many words start with each letter.

Create a LINQ query to find customers who placed orders in the last month, sorted by order date.

Write a query to calculate the total quantity of each product sold from an order detail collection.

### Practice 3

## LINQ Methods

Use Select to transform a list of employee objects into a list of their email addresses.

Use FirstOrDefault to find the first student with a grade above 90, or return null if none exist.

Implement a LINQ chain using Select and Where to find the names of people aged above 30.

Use Aggregate to find the concatenated string of all book titles in a collection, separated by commas.

Use Take and Skip to implement pagination for displaying 5 records per page from a large dataset.

#### Practice 4

### Query Operators

#### Where

Write a query to filter all employees with a salary greater than \$50,000.

Use Where to retrieve all even numbers from a list of integers.

Write a LINQ query to find all products that are both in stock and cost less than \$20.

Use Where to filter students whose names start with a specific letter provided by the user.

Retrieve all records from a dataset where a specific field (e.g., age) satisfies a dynamic condition.

#### Practice 5

### OfType

Use OfType<T> to filter only integer elements from a mixed collection.

Write a query to find all strings from an ArrayList of mixed data types.

Use OfType to filter all floating-point numbers from a collection and calculate their average.

Implement a LINQ query to extract all objects of a specific type (e.g., Product) from a collection of objects.

Retrieve only DateTime elements from a list containing various data types.

#### Practice 6

### OrderBy

Write a query to sort a list of students by their grades in descending order.

Use OrderBy to arrange employee names alphabetically.

Sort a list of products first by category and then by price within each category using OrderBy and ThenBy.

Implement a LINQ query to sort a collection of strings by their length.

Write a LINQ query to sort a dataset by multiple fields, dynamically chosen at runtime.

#### Practice 7

### GroupBy, ToLookup

Use GroupBy to group a list of employees by their department and count the number of employees in each department.

Write a LINQ query to group a collection of products by their categories and calculate the total stock for each category.

Implement a query using ToLookup to group customers by their country and retrieve all customers from a specific country.

Use GroupBy to group a list of words by their length and sort the groups by the length of the words.

Implement a LINQ query to group orders by their status and find the total revenue for each status.

#### Practice 8

### Join

Write a LINQ query to join two lists: employees and departments, and display employee names along with their department names.

Use Join to combine a list of orders with a list of customers based on customer ID.  
 Implement a query to join a list of students with their grades and display the student's name and grade.  
 Use Join to create a combined list of products and their suppliers based on a common field.  
 Write a LINQ query to join two lists and filter the result by a specific condition (e.g., orders placed after a specific date).

## Practice 9

### GroupJoin

Use GroupJoin to group products by their categories and display the category name along with its products.  
 Write a query to group customers with their orders using GroupJoin, displaying the customer name and their orders.  
 Implement a LINQ query to group a list of teachers with their students based on a TeacherId.  
 Use GroupJoin to combine employees with their projects and calculate the number of projects each employee is handling.  
 Write a LINQ query to group departments with their employees and sort the departments by the number of employees.

## Assignment Exercise

### Assignment 1

Create a Two Classes Employees and Incentives class Employee { public int ID { get; set; } public string FirstName { get; set; } public string LastName { get; set; } public double Salary { get; set; } public DateTime JoiningDate { get; set; } public string Department { get; set; } } class Incentive { public int ID { get; set; } public double IncentiveAmount { get; set; } public DateTime IncentiveDate { get; set; } } class Program { static void Main(string[] args) { List employees = new List() { new Employee() { ID=1,FirstName="John",LastName="Abraham",Salary=1000000,JoiningDate=new DateTime(2013,1,1),Department="Banking"}, new Employee() { ID=2,FirstName="Michael",LastName="Clarke",Salary=800000,JoiningDate=new DateTime(),Department="Insurance" }, new Employee() { ID=3,FirstName="oy",LastName="Thomas",Salary=700000,JoiningDate=new DateTime(),Department="Insurance"}, new Employee() { ID=4,FirstName="Tom",LastName="Jose",Salary=600000,JoiningDate=new DateTime(),Department="Banking"}, new Employee() { ID=4,FirstName="TestName2",LastName="Lname%",Salary=600000,JoiningDate=new DateTime(2013,1,1),Department="Services" } }; List incentives = new List() { new Incentive() { ID=1,IncentiveAmount=5000,IncentiveDate=new DateTime(2013,02,02)}, new Incentive() { ID=2,IncentiveAmount=10000,IncentiveDate=new DateTime(2013,02,4)}, new Incentive() { ID=1,IncentiveAmount=6000,IncentiveDate=new DateTime(2012,01,5)}, new Incentive() { ID=2,IncentiveAmount=3000,IncentiveDate=new DateTime(2012,01,5)} }; } } Solve below queries:

### Assignment 2

Get employee details from employees object whose employee name is "John" (note restrict operator)

### Assignment 3

Get FIRSTNAME, LASTNAME from employees object (note project operator)

#### Assignment 4

Select FirstName, IncentiveAmount from employees and incentives object for those employees who have incentives. (join operator)

#### Assignment 5

Get department wise maximum salary from employee table order by salary ascending (note group by)

#### Assignment 6

Select department, total salary with respect to a department from employees object where total salary greater than 800000 order by TotalSalary descending (group by having)

#### Assignment 7

### Scenario

You are tasked with developing a **Student Management System** for a college. The system will manage data about students, courses, and enrollments. It will allow administrators to query and analyze data using LINQ.

## Requirements

### Data Models

Create the following classes:

#### Student

StudentId (int)  
Name (string)  
Age (int)  
Gender (string)  
Department (string)

#### Course

CourseId (int)  
CourseName (string)  
Department (string)

#### Enrollment

EnrollmentId (int)  
StudentId (int)  
CourseId (int)  
EnrollmentDate (DateTime)

## Tasks

### Task 1: Filter and Analyze Data (Where and OfType)

Retrieve all students who belong to the "Computer Science" department and are older than 20 years.

Filter and display all course names from a mixed collection of objects using OfType<string>.

### Task 2: Sorting and Grouping (OrderBy, GroupBy, ToLookup)

Sort all students by their name in ascending order. Then, sort them by age in descending order.

Group all students by their department and display the total count of students in each department.

Use ToLookup to group courses by department and retrieve all courses belonging to a specific department (e.g., "Mathematics").

### Task 3: Joining Data (Join and GroupJoin)

Write a LINQ query to join students with their enrollments and display the student name along with the course name they are enrolled in.

Use GroupJoin to group students with the courses they are enrolled in and display the student name along with the list of course names.

### Task 4: Advanced Queries (LINQ Query and Methods)

Calculate the total number of enrollments in each department and display the department name along with the count.

Find the student(s) with the maximum number of enrollments.

Write a LINQ query to find all courses that have no enrollments.

Student Management System

-----

1. View Students in a Department
2. View Courses by Department
3. View Enrollments (Student and Course)
4. Find Students with Maximum Enrollments
5. View Courses with No Enrollments
6. Exit

Online Reference

<https://www.tutorialsteacher.com/linq/linq-lambda-expression>

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 7

## What to learn

### LINQ

Select

All/Any

Contains

Aggregate

Average

Count

Single/SingleOrDefault

DefaultIfEmpty

Intersect/Union/Take,TakeWhile

Deferred Execution

Immediate Execution

Let Keyword

Into Keyword

## Practice Exercise

### Practice 1

#### Select

Use Select to extract only the names of employees from a list of Employee objects and display them.

Write a LINQ query to create a new list of Student objects with only Name and Age properties from the original collection.

Use Select to convert a list of product prices from USD to EUR using a conversion rate.

Write a LINQ query to transform a list of dates into their corresponding day names (e.g., "Monday", "Tuesday").

Use Select to create a list of email addresses from a collection of user objects.

### Practice 2

#### All/Any

Use All to check if all products in a list are in stock.

Use Any to verify if there is at least one student with a grade above 90 in a list of scores.

Write a LINQ query to check if all employees in a department have salaries above a certain threshold.

Use Any to determine if there are pending orders in an order list with a status of "Pending".

Write a query to check if all courses have at least one student enrolled.

### Practice 3

#### Contains

Use Contains to check if a specific product ID exists in a list of product IDs.



Write a LINQ query to determine if a list of names contains a specific name entered by the user.

Use Contains to filter orders that match specific order IDs from a predefined list.

Write a query to check if a list of books includes a specific title using Contains.

Use Contains to find if a list of countries includes a specific country entered by the user

#### Practice 4

### Aggregate

Use Aggregate to concatenate all product names from a list into a single comma-separated string.

Write a query to calculate the product of all numbers in a list using Aggregate.

Use Aggregate to find the longest word in a list of strings.

Implement a LINQ query to compute the cumulative total of sales from a list of transactions using Aggregate.

Use Aggregate to reverse a string by iterating through its characters.

#### Practice 5

### Average

Use Average to calculate the average age of employees in a department.

Write a LINQ query to find the average grade of students in a class.

Calculate the average price of all products in a catalog using LINQ.

Write a query to compute the average transaction amount from a list of sales.

Use LINQ to find the average word length in a collection of sentences.

#### Practice 6

### Count

Use Count to find the number of products in a list with a price greater than \$50.

Write a LINQ query to count the number of students enrolled in a specific course.

Use Count to find how many numbers in a list are divisible by 3.

Implement a query to count the number of orders placed in the last 7 days.

Use Count to determine how many employees in a company belong to a specific department.

#### Practice 7

### Single/SingleOrDefault

Use Single to find the product with a unique ID from a product list. Throw an exception if not found.

Use SingleOrDefault to retrieve a user from a list by their unique username. Return null if no match is found.

Write a query to find the student with a unique roll number from a list of students.

Use SingleOrDefault to retrieve a transaction from a list with a specific transaction ID.

Ensure the code handles cases with no match.

Use Single to find the book with a unique ISBN from a catalog.

#### Practice 8

### DefaultIfEmpty

Use DefaultIfEmpty to handle cases where a query to find high-priority tasks returns no results.

Write a LINQ query to find products in a specific category. Return a default message if the category is empty.

Use `DefaultIfEmpty` to ensure a collection of employees in a department always returns at least one placeholder.

Write a LINQ query to retrieve all orders for a specific date and return "No Orders Found" if none exist.

Use `DefaultIfEmpty` to provide a default value when filtering a list of customers.

#### Practice 9

### Intersect/Union/Take/TakeWhile

Use `Union` to combine two lists of integers, removing duplicates.

Use `Intersect` to find common employees between two departments.

Write a query using `Take` to retrieve the first 5 students from a list of grades.

Use `TakeWhile` to retrieve scores from a list until a score below 50 is encountered.

Use `Union` to merge two lists of courses and eliminate duplicates.

#### Practice 10

### Deferred Execution

Write a LINQ query to demonstrate deferred execution by creating a query and executing it only after modifying the source collection.

Implement a program that demonstrates deferred execution using a list of integers and a `Where` clause.

Show how deferred execution works by iterating through a collection after adding new elements to it.

Write a query that retrieves all even numbers from a list and demonstrate deferred execution by modifying the list after the query is defined.

Use deferred execution to filter a collection and only execute the filter when iterated.

#### Practice 11

### Immediate Execution

Use `ToList` to force immediate execution of a query that filters products by price.

Write a LINQ query that calculates the sum of a list of numbers using immediate execution.

Demonstrate immediate execution by converting a query result to an array using `ToArray`.

Use `ToDictionary` to immediately execute a query and store the result as a dictionary.

Compare deferred and immediate execution using a query that uses `ToList`.

#### Practice 12

### Let Keyword

Use the `let` keyword to store the result of a calculation (e.g., square of a number) in a LINQ query.

Write a query that uses `let` to define an intermediate variable for the length of strings and filter strings by their lengths.

Use `let` to calculate the discounted price of products in a catalog and filter products with discounts above \$10.

Write a LINQ query to split names into first and last names using `let` and filter by last name length.

Use `let` to simplify a LINQ query that filters a collection based on a complex condition.

## Practice 13

### Into Keyword

Use the into keyword to create a continuation query that filters employees by department and then sorts them by age.

Write a query that groups students by their grades and continues with an additional filter using into.

Use into to chain queries that first select products by category and then sort them by price.

Write a LINQ query that uses into to filter words by length and then group them by their first letter.

Implement a program that uses into to refine a group of customers by country and filter those with more than 5 orders.

## Assignment Exercise

### Assignment 1

#### Objective:

This assignment focuses on utilizing LINQ concepts such as **Select**, **All/Any**, **Contains**, **Aggregate**, **Average**, **Count**, **Single/SingleOrDefault**, **DefaultIfEmpty**, **Intersect/Union/Take/TakeWhile**, **Deferred Execution**, **Immediate Execution**, **Let Keyword**, and **Into Keyword** to solve real-world scenarios effectively.

### Scenario

You are tasked with creating a **Business Analytics Tool** for a retail company. The tool will analyze customer data, sales data, and product inventory using LINQ to provide actionable insights.

## Requirements

### Data Models

Create the following classes:

#### Customer

CustomerId (int)  
Name (string)  
City (string)  
IsPremium (bool)

#### Product

ProductId (int)  
ProductName (string)  
Category (string)  
Price (decimal)  
Stock (int)

#### Order

OrderId (int)  
CustomerId (int)

ProductId (int)  
Quantity (int)  
OrderDate (DateTime)

## Tasks

### Task 1: Basic LINQ Queries

Use Select to create a list of product names and their prices.  
Use All to check if all customers are from the same city.  
Use Any to verify if there is at least one premium customer in the dataset.  
Use Contains to check if a specific product exists in the product list by name.  
Use Aggregate to concatenate all customer names into a single string, separated by commas.

### Task 2: Statistical Analysis

Use Average to calculate the average price of products in a specific category.  
Use Count to find how many orders were placed in the last 30 days.  
Use SingleOrDefault to retrieve a product with a specific product ID, and handle cases where the product is not found.  
Use DefaultIfEmpty to handle a scenario where there are no orders for a specific date. Return a default message if the list is empty.  
Use Take to retrieve the first 5 customers from the customer list.

### Task 3: Combining and Filtering Data

Use Intersect to find common products sold between two specific categories.  
Use Union to merge two lists of products from different warehouses, eliminating duplicates.  
Use TakeWhile to retrieve orders from a list until an order with quantity less than 5 is encountered.  
Demonstrate **Deferred Execution** by modifying the product list after defining a LINQ query and observe the impact on the query results.  
Demonstrate **Immediate Execution** by converting a query result to a list using ToList.

### Task 4: Complex Queries

Use the let keyword to calculate the total price (price × quantity) for each order and filter orders with a total price above \$100.  
Use the into keyword to group customers by their city and filter cities with more than 5 customers.  
Write a LINQ query to group products by category, calculate the average price per category, and display the result.  
Use let to define an intermediate variable for stock status (e.g., "Low Stock" if stock < 10) and filter products based on this status.  
Use into to create a continuation query that first filters premium customers and then sorts them by their names.

#### Online Reference

<https://www.tutorialsteacher.com/linq/linq-lambda-expression>

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 8

## What to learn

Async and await  
Extension Method

## Practice Exercise

### Practice 1

Async and await

### Practice 2

Do one example of async await

### Practice 3

Do one example of Extension Method

### Practice 4

## Async and Await

### Asynchronous File Download

Write a method that uses async and await to download a file from a given URL and save it locally. Ensure that the method can handle the download asynchronously and provide feedback on the download progress.

### Asynchronous Database Query

Implement an async method to query customer details from a database. The method should simulate a database delay using Task.Delay and return the customer data asynchronously.

### Parallel API Calls

Write an async method that makes parallel API calls to fetch product details from three different online stores. Use await for each request and ensure that all data is returned in a consolidated format once all calls are completed.

### Asynchronous Order Processing

Simulate an e-commerce system where order processing involves multiple asynchronous operations, such as checking stock, calculating taxes, and applying discounts. Write an async method that processes the order and returns a confirmation asynchronously.

### Async File Processing

Write an async method to process a large number of files asynchronously. The method should read the files, process their content, and save the results back to a

storage system. Ensure that the method completes each file's processing asynchronously.

## Practice 5

# Extension Method

### Custom String Validation

Implement an extension method `IsValidEmail()` for the string type that validates if an email address is in a valid format (basic validation). Use this extension to check whether a list of email addresses is valid.

### Collection Average Calculation

Create an extension method `AveragePrice()` for a `List<Product>` class, where each `Product` contains a `Price` property. Use this method to calculate the average price of all products in a list.

### String Reversal

Implement an extension method `ReverseString()` for the string class that returns the reversed version of a given string. Use this method to reverse a list of strings and output the results.

### Custom Sorting

Create an extension method `SortByName()` for a `List<Employee>` class that sorts employees by their `Name` property. Implement a small program that uses this extension method to sort and display employees.

### Date Formatting

Implement an extension method `ToFormattedDate()` for `DateTime` that formats the date into a custom format (e.g., "yyyy-MM-dd"). Write a program to display a list of dates in this formatted version.

## Assignment Exercise

### Assignment 1

#### Objective:

This assignment combines the concepts of **Async and Await** and **Extension Methods** to create a real-world business application that processes orders, performs asynchronous operations, and uses extension methods to extend the functionality of existing types.

## Scenario: E-Commerce Order Processing System

You are building an **E-Commerce System** that processes customer orders. The system needs to handle the following tasks:

**Order Validation:** Check if the product is in stock.

**Tax Calculation:** Asynchronously calculate taxes based on the product price.

**Discount Application:** Asynchronously apply discounts for premium customers.

**Shipping:** Use async operations to simulate different shipping methods.

**Order Summary:** Extend existing types to format and output order details.

You will implement the following features:

## Requirements

### Data Models

Create the following classes to represent the system:

#### Customer

CustomerId (int)  
Name (string)  
IsPremium (bool)

#### Product

ProductId (int)  
Name (string)  
Price (decimal)  
Stock (int)

#### Order

OrderId (int)  
CustomerId (int)  
ProductId (int)  
Quantity (int)  
OrderDate (DateTime)

#### ShippingInfo

ShippingId (int)  
ShippingMethod (string)  
ShippingCost (decimal)

## Tasks

### 1. Asynchronous Order Processing

Write an async method `ProcessOrderAsync(Order order)` that:

Validates if the product is in stock.

Simulates tax calculation with a delay.

Applies a discount for premium customers (if applicable).



Determines the shipping cost based on the shipping method chosen.

## 2. Extension Methods

### Product Extension:

Create an extension method `IsInStock(this Product product)` for the `Product` class that checks if the quantity of a product is greater than zero.

### Order Extension:

Create an extension method `CalculateTotal(this Order order, Product product)` that calculates the total price for the order ( $\text{price} * \text{quantity}$ ) and applies any applicable discount for premium customers.

### Shipping Extension:

Create an extension method `ApplyShippingCost(this ShippingInfo shippingInfo)` that calculates the shipping cost based on the shipping method.

### Date Extension:

Create an extension method `ToFormattedDate(this DateTime date)` for the `DateTime` class to return a custom formatted date string (e.g., "yyyy-MM-dd").

## 3. Main Program Logic

Implement a main program that:

- Creates a list of customers, products, and orders.

- Calls the `ProcessOrderAsync` method for each order and outputs the order summary after processing.

- Use extension methods for calculating totals, checking stock, and applying shipping costs.

- Displays the order details with formatted dates.

Online Reference

No online Reference

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 9

## What to learn

- Learn File Handling
- File Class
- File Stream
- Binary Reader
- Binary Writer
- Stream Reader
- Stream Writer
- FileInfo
- Directory Info
- DriveInfo
- Path
- Converting object to json

## Practice Exercise

### Practice 1

<https://learn.microsoft.com/en-us/dotnet/api/system.io.file?view=net-7.0>

### Practice 2

<https://learn.microsoft.com/en-us/dotnet/api/system.io.fileinfo?view=net-7.0>

### Practice 3

<https://learn.microsoft.com/en-us/dotnet/api/system.io.directoryinfo?view=net-7.0>

### Practice 4

<https://www.tutorialsteacher.com/articles/convert-object-to-json-in-csharp>

### Practice 5

## File Class

**Create a text file:** Write a C# program to create a text file named output.txt and write some sample content into it using the File class.

**Read from a file:** Given a text file sample.txt, write a C# program to read the content of the file using the File class and display it on the console.

**Check if a file exists:** Write a C# program to check whether a file named data.txt exists in the current directory and output a message accordingly.

**Delete a file:** Write a C# program to delete a file named deleteMe.txt from the current directory using the File class.

**Copy a file:** Create a program that copies the file source.txt to destination.txt using the File.Copy method.

## Practice 6

# File Stream

**Write to a file using FileStream:** Write a C# program to create a FileStream that writes the string "Hello World!" to a file named output.txt.

**Read from a file using FileStream:** Write a C# program that reads from the file input.txt and displays its content on the console using a FileStream.

**Append to a file using FileStream:** Write a C# program that appends the string "Appended Text" to the file example.txt using the FileStream class.

**FileStream for binary data:** Write a C# program that writes an array of bytes to a file named binaryData.dat using a FileStream.

**Using FileStream to check file length:** Write a C# program to open a file data.txt using a FileStream and print its length in bytes.

## Practice 7

# Binary Reader

**Read a number using BinaryReader:** Write a C# program to read an integer stored in a binary file data.dat using a BinaryReader.

**Read a string using BinaryReader:** Write a C# program that reads a string from a binary file sample.dat using the BinaryReader class and prints it.

**Read multiple types from a binary file:** Create a program that uses BinaryReader to read an integer, a double, and a string from a binary file and prints each one.

**Handling end of file with BinaryReader:** Write a C# program that reads from a binary file data.dat using BinaryReader and stops reading when the end of the file is reached.

**BinaryReader for reading a float:** Write a program to read a floating-point number from a binary file floatData.dat using BinaryReader and display it.

## Practice 8

# Binary Writer

**Write an integer using BinaryWriter:** Write a C# program that writes an integer 1234 to a binary file data.dat using BinaryWriter.

**Write a string to a binary file:** Write a C# program that writes the string "Hello Binary" to a binary file message.dat using BinaryWriter.

**Write multiple data types using BinaryWriter:** Write a program that uses BinaryWriter to write an integer, a double, and a string to a binary file.

**Write a float to a binary file:** Create a C# program that writes a floating-point number 3.14 to a binary file floatData.dat using BinaryWriter.

**Append data with BinaryWriter:** Write a C# program that appends the string "Appended text" to a binary file appendData.dat using BinaryWriter.

## Practice 9

# Stream Reader

**Read from a text file using StreamReader:** Write a C# program to read the content of a text file input.txt using StreamReader and display it on the console.

**Read a line using StreamReader:** Write a C# program that reads a single line from the file data.txt using StreamReader.

**Read all lines using StreamReader:** Write a C# program that reads all the lines from the file lines.txt using StreamReader and displays them.

**StreamReader to check for end of file:** Write a program that reads from file.txt using StreamReader and stops when the end of the file is reached.

**Read a file with StreamReader and count words:** Write a program that uses StreamReader to read a text file textfile.txt and counts the number of words in the file.

## Practice 10

# Stream Writer

**Write to a file using StreamWriter:** Write a C# program that writes "Welcome to C#" to a text file greeting.txt using StreamWriter.

**Append text to a file using StreamWriter:** Write a program that appends the text "Appended Text" to an existing file file.txt using StreamWriter.

**Write multiple lines using StreamWriter:** Write a C# program that writes multiple lines (e.g., "Line 1", "Line 2") to a text file lines.txt using StreamWriter.

**Write formatted data using StreamWriter:** Write a program that uses StreamWriter to write a formatted string, such as "Name: John, Age: 30", to a file.

**Write and flush with StreamWriter:** Write a program that writes some text to a file and explicitly calls `StreamWriter.Flush()` to ensure the content is written to the file.

## Practice 11

### FileInfo

**Get file size using FileInfo:** Write a C# program that uses `FileInfo` to get the size of the file `data.txt` and displays it.

**Check if a file exists using FileInfo:** Write a program that uses `FileInfo` to check whether the file `config.txt` exists and displays a message.

**Get the creation time of a file:** Write a C# program that uses `FileInfo` to get and display the creation time of a file `file.txt`.

**Rename a file using FileInfo:** Write a program that uses `FileInfo` to rename a file `oldname.txt` to `newname.txt`.

**Delete a file using FileInfo:** Write a program that uses `FileInfo` to delete a file `unwantedfile.txt`.

## Practice 12

### DirectoryInfo

**Get the list of files in a directory:** Write a program that uses `DirectoryInfo` to get all the file names in a directory `C:\files` and display them.

**Create a directory using DirectoryInfo:** Write a C# program that creates a new directory `C:\newFolder` using `DirectoryInfo`.

**Get directory creation time using DirectoryInfo:** Write a program that uses `DirectoryInfo` to get and display the creation time of a directory.

**Check if a directory exists using DirectoryInfo:** Write a program that checks whether the directory `C:\test` exists using `DirectoryInfo` and displays a message.

**Delete a directory using DirectoryInfo:** Write a program that deletes the directory `C:\tempDir` using `DirectoryInfo`.

## Practice 13

### DriveInfo

**Get available free space on a drive:** Write a program that uses `DriveInfo` to get the available free space on the `C:` drive.

**Get drive type using DriveInfo:** Write a program that uses `DriveInfo` to determine if the `C:` drive is a fixed drive or a removable drive.

**Get total size of a drive using DriveInfo:** Write a C# program that uses DriveInfo to get and display the total size of the C: drive.

**Get drive name using DriveInfo:** Write a program that uses DriveInfo to get and display the name of the C: drive.

**Check if a drive is ready using DriveInfo:** Write a program that checks if the D: drive is ready and displays a message accordingly.

#### Practice 14

## Path

**Get the file name from a path:** Write a C# program that extracts and displays the file name from the path C:\files\example.txt using the Path class.

**Get the directory name from a path:** Write a program that extracts and displays the directory name from the path C:\files\example.txt using the Path class.

**Combine two paths using Path.Combine:** Write a program that combines the directory path C:\files and the file name data.txt using Path.Combine.

**Get the file extension using Path:** Write a program that extracts and displays the file extension from the path C:\files\document.pdf using Path.GetExtension.

**Check if a path is rooted using Path:** Write a program that checks if the path C:\files\example.txt is rooted using Path.IsPathRooted.

#### Practice 15

## Converting Object to JSON

**Convert an object to JSON:** Write a C# program that converts a Person object with properties Name and Age to a JSON string using JsonConvert.SerializeObject().

**Serialize a list of objects to JSON:** Write a program that serializes a list of Product objects to a JSON array using JsonConvert.SerializeObject().

**Deserialize JSON to an object:** Write a C# program that deserializes a JSON string representing a Person object back to a Person class.

**Handle JSON array deserialization:** Write a program that deserializes a JSON array string into a list of Product objects.

**JSON conversion with nested objects:** Write a program that converts an object containing nested objects (e.g., an Order object with a list of Items) into a JSON string.

## Assignment 1

```
[
  {
    "MovieID": 1,
    "MovieName": "Pathan",
    "Details": {
      "DirectorName": "Sidharth Anand",
      "ActorsNames": [
        "Shahrukh Khan",
        "Deepika Padukone",
        "John Abraham"
      ],
      "VideoLink": "https://tutorial.radixind.in/videos/Pathaan.mp4"
    }
  },
  {
    "MovieID": 2,
    "MovieName": "Dabang 3",
    "Details": {
      "DirectorName": "Prabhu Deva",
      "ActorsNames": [
        "Sonakshi Sinha",
        "Salman Khan"
      ],
      "VideoLink": "https://tutorial.radixind.in/videos/KKBKKJ.mp4"
    }
  },
]
```

```
{  
  "MovieID": 3,  
  "MovieName": "Kisi Ka bhai Kisi ki Jaan",  
  "Details": {  
    "DirectorName": "Farhad Samji",  
    "ActorsNames": [  
      "Pooja Hegde",  
      "Salman Khan"  
    ],  
    "VideoLink": "https://tutorial.radixind.in/videos/Dabaang3.mp4"  
  }  
},  
{  
  "MovieID": 4,  
  "MovieName": "Any Body can Dance",  
  "Details": {  
    "DirectorName": "Prabhu Deva",  
    "ActorsNames": [  
      "Prabhu Deva",  
      "Punit Pathak",  
      "Ganesh Acharya",  
      "Noorin sha"  
    ],  
    "VideoLink": "https://tutorial.radixind.in/videos/abcd.mp4"  
  }  
}
```



```
]
```

store above data in movie.json file in your project folder.

```
{
```

```
  "Directors": [
```

```
    "Prabhu Deva",
```

```
    "Sidharth Anand",
```

```
    "Farhad Samji"
```

```
  ],
```

```
  "Movies": [
```

```
    {
```

```
      "MovieID": 1,
```

```
      "MovieName": "Pathan"
```

```
    },
```

```
    {
```

```
      "MovieID": 2,
```

```
      "MovieName": "Dabang 3"
```

```
    },
```

```
    {
```

```
      "MovieID": 3,
```

```
      "MovieName": "Kisi Ka bhai Kisi ki Jaan"
```

```
    },
```

```
    {
```

```
      "MovieID": 4,
```

```
      "MovieName": "Any Body can Dance"
```

```
    }
```

```
  ],
```

```
"Actors": [  
    "Shahrukh Khan",  
    "Deepika Padukone",  
    "John Abraham",  
    "Sonakshi Sinha",  
    "Salman Khan",  
    "Pooja Hegde",  
    "Salman Khan",  
    "Prabhu Deva",  
    "Punit Pathak",  
    "Ganesh Acharya",  
    "Noorin sha"  
]  
}
```

store this info.json file in your project folder.

Task:

List all the director to the user.

Create classes and interfaces accordingly.

Input a director name from the user.

based on the director name its movies should be displayed

Show all the movies name and take input movie name from the user and displays its actors name.

Print that movie name where director is working as a actor in the movie.

## Assignment 2

Display folder hierarchy of any of folder. Recursion should be used

example output

A

B

```

C
D
    E
    F
        info.txt
        hell.txt
p.txt
```

### Assignment 3

There is one image gallery folder in your system. keep only that images which is modified in the current month other files should be deleted

### Assignment 4

#### Scenario: File Management System for a Company

You are tasked with creating a simple file management system for a company that keeps track of employee records and the documents they work with. The system should be able to read, write, and modify files, handle directories, and store employee information in a structured format. You will also implement serialization and deserialization of data to store employee records in JSON format for future use.

#### Requirements:

##### File and Directory Operations:

Create a directory structure for storing employee files. The main directory will be `C:\CompanyFiles\Employees\`, where each employee will have a folder named after their Employee ID. For example:

`C:\CompanyFiles\Employees\12345\`.

In each employee folder, create a text file called `info.txt` containing the employee's details (Name, Age, Department, Position).

Use the `File` and `DirectoryInfo` classes to check if the employee folder exists. If it does not, create the folder and the `info.txt` file with sample data.

##### Stream Operations (Reading and Writing Files):

Implement a function that reads the employee's details from the `info.txt` file in their respective folder using `StreamReader`. If the file does not exist, handle the error by displaying an appropriate message.

Implement a function to update the `info.txt` file with new information using `StreamWriter`. The program should ask for the new data (e.g., new position) and update the text file.

##### File Info and Drive Info:

Using `FileInfo`, get and display the size of the `info.txt` file for a specific employee.

Use DriveInfo to check if there is enough free space in the drive (C:) for creating new employee folders (set a threshold of 100MB).

### **JSON Serialization and Deserialization:**

Create a class Employee with properties EmployeeId, Name, Age, Department, and Position. Serialize an instance of this class to a JSON string using JsonConvert.SerializeObject() and store it in a file named employee.json inside the employee folder.

Implement a function to deserialize the employee.json file and load the employee data into the Employee object.

### **Additional Operations:**

Use Path class methods to determine the file extension of info.txt and print it to the console.

Implement a function that appends a document name (e.g., resume.pdf) to a documents.txt file inside the employee folder using StreamWriter (if documents.txt does not exist, create it).

### **Bonus: File Management System Operations**

Implement the functionality to display all employee folders in the C:\CompanyFiles\Employees\ directory.

Write a method that takes an employee's ID and retrieves the document list from documents.txt, displaying the names of all files associated with that employee.

### **Online Reference**

No online Reference

### **.NET Core Web API**

#### **WEB API (old)**

#### **Authentication And Authorization (WEBAPI)(old)**

#### **FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 10

## What to learn

Entity Framework Core  
What is Entity Framework Core  
DB context Methods  
Database First Approach  
Include method

## Practice Exercise

### Practice 1

Do the practice exercise from msdn for given topics

### Practice 2

Create a new DbContext class named AppDbContext with a DbSet for the Product entity.  
Configure a connection string in AppDbContext for a SQL Server database named ShopDb.  
Install the required NuGet packages for using Entity Framework Core in a .NET Core project. Write the installation commands.  
Add a new product with the following details to the database: Name = "Phone", Price = 500.  
Retrieve all products where the price is greater than 100 using LINQ.  
Update the name of the product with Id = 1 to "Smartphone."

### Practice 3

## Migrations

Enable migrations for a new Entity Framework Core project. Write the command used to initialize migrations.

Create a migration to add a new column Description to the Product entity.

Apply the migration to the database and verify that the new column is created.

### Practice 4

Create a one-to-many relationship between Category and Product. Add navigation properties and update the model.

Seed initial data for Category and Product tables in the OnModelCreating method.

Use the Include method to retrieve all products along with their respective categories.

```
public class Category {    public int Id { get; set; }    public string Name { get; set; }    public List<Product> Products { get; set; } }
```

## Practice 5

### Fluent API

Configure the Product entity to make the Name column required and set a maximum length of 100 using Fluent API.

Create a composite key for an entity named OrderDetail with OrderId and ProductId.

Use Fluent API to specify that the Price column in the Product entity should have a precision of 10 and scale of 2.

## Assignment Exercise

### Assignment 1

Create A Database design and insert record in the database. Create views In a hospital there are different departments. Patients are treated in these departments by the doctors assigned to patients. Usually each patient is treated by a single doctor, but in rare cases they will have two or three. Healthcare assistants will also attend to patients; every department has many healthcare assistants. Each patient is required to take a variety of drugs during different parts of the day such as morning, afternoon and night.

### Assignment 2

After Creating a Database design. Create ORM with database first Approach. Insert a Doctor Update a Doctor Delete a Doctor Find a report of patient assigned to a particular doctor Find a report of medicine list for a particular patient Display summary report of Doctor and patient (use Include method)

## Online Reference

<https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx>

<https://www.entityframeworktutorial.net/efcore/install-entity-framework-core.aspx>

<https://www.entityframeworktutorial.net/efcore/create-model-for-existing-database-in-ef-core.aspx>

## .NET Core Web API

### WEB API (old)

### Authentication And Authorization (WEBAPI)(old)

### FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 11

## What to learn

Code First Approach Entity Framework

One-to-One Relationship

One-to-many Relationship

One-to-many Relation

Fluent API

One-to-One Relationship

One-to-many Relationship

One-to-many Relation

Migration Commands

Add-Migration

Remove-Migration

Update-Database

Script-DbContext

Get-DbContext

Drop-Database

## Practice Exercise

### Practice 1

Do the exercise from MSDN for given topics

## Assignment Exercise

### Assignment 1

A toy manufacturing company manufactures different types of toys. The company has several manufacturing plants. Each plant manufactures different types of toys. A customer can place the order for these toys. Each order may contain one or more toys. Each customer has multiple ship-to addresses. To promote the business, the company offers different schemes based on the order value. use Store procedure

### Assignment 2

Define Relationship and do crud operation customer, customer can view all the products and search the record and place an order.

### Assignment 3

**Duration:** 2 Hours

**Objective:** Practice core Entity Framework concepts, including setup, CRUD operations, relationships, migrations, Fluent API configurations, and advanced querying.

## Assignment Details

### Scenario:

You are building a small database for an online store with the following requirements:

The store sells **Products**, each belonging to a **Category**.

Customers can place **Orders**, and each order contains multiple **Products**.

The database must support common operations such as adding, updating, retrieving, and deleting records.

## Tasks

### 1. Setting Up Entity Framework (20 minutes)

Create a new console or ASP.NET Core application.

Add the required NuGet packages for Entity Framework Core and SQL Server.

Create the following entities:

Create a DbContext class named AppDbContext with DbSet properties for the above entities.

Configure a SQL Server connection string in AppDbContext.

### 2. Migrations and Seeding (20 minutes)

Enable migrations and create an initial migration.

Apply the migration to create the database.

Seed the database with the following initial data:

**Categories:** Electronics, Furniture.

**Products:**

Laptop (Price: 1500, Category: Electronics).

Sofa (Price: 800, Category: Furniture).

### 3. CRUD Operations (20 minutes)

Add a new product: **Phone** (Price: 700, Category: Electronics).

Retrieve and display all products in the database.

Update the price of the **Laptop** to 1600.

Delete the **Sofa** product from the database.

### 4. Relationships and Queries (30 minutes)



Implement a one-to-many relationship between Category and Product (already modeled in the entities).

Add a product to an order using the Order entity.

Write a query to retrieve all products along with their categories using the Include method.

Write a query to retrieve the top 2 most expensive products.

Write a query to count the number of products in each category.

## 5. Fluent API and Advanced Configuration (30 minutes)

Use Fluent API to configure the Name property of Product to be required and have a maximum length of 100.

Configure a composite key for the Order entity using Id and OrderDate.

Specify that the Price column in Product should have a precision of 10 and a scale of 2 using Fluent API.

csharp

Copy code

```
public class Product {    public int Id { get; set; }    public string Name { get; set; }    public decimal Price { get; set; }    public int CategoryId { get; set; }    public Category Category { get; set; } }    public class Category {    public int Id { get; set; }    public string Name { get; set; }    public List<Product> Products { get; set; } }    public class Order {    public int Id { get; set; }    public DateTime OrderDate { get; set; }    public List<Product> Products { get; set; } }
```

Online Reference

No online Reference

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 12

## What to learn

Code First Approach Entity Framework

One-to-One Relationship

One-to-many Relationship

One-to-many Relation

Fluent API

One-to-One Relationship

One-to-many Relationship

One-to-many Relation

Migration Commands

Add-Migration

Remove-Migration

Update-Database

Script-DbContext

Get-DbContext

Drop-Database

## Practice Exercise

No practice exercise

## Assignment Exercise

No assignment exercise

## Online Reference

No online Reference

## .NET Core Web API

### WEB API (old)

### Authentication And Authorization (WEBAPI)(old)

### FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 13

## What to learn

.NET Core vs. .NET Framework

Key features of .NET Core

Setting up the development environment (Visual Studio/VS Code)

Introduction to RESTful APIs

Creating HTTP methods: GET, POST, PUT, DELETE

Retrieve Headers Information

Creating a new .NET Core Web API project

Hands-on: Build a basic "Hello World" Web API.

Hands-on: Build a in Memory .NET Core WebAPI with  
GET/PUT/POST/PATCH/DELETE

## Practice Exercise

### Practice 1

Create PPT and Explain learned Topics in this module

### Practice 2

Create WebAPI for day13 task

### Practice 3

**Exercise: Building a Simple API**

Create a RESTful API with one endpoint /greet that returns "Hello, World!".

**Business Logic:** Extend it to accept a query parameter name and return "Hello, [Name]!".

**Outcome:** Understand the basics of creating an API.

### Practice 4

**Exercise: Statelessness in APIs**

Create a RESTful API that calculates the factorial of a number passed as a query parameter (e.g., /factorial?number=5).

**Business Logic:** Return a 400 Bad Request if the number is not a positive integer.

**Outcome:** Understand stateless behavior of REST APIs.

### Practice 5

## Exercise: REST Constraints

Create a RESTful API with two endpoints:

`/uppercase?text=example` to convert the text to uppercase.

`/reverse?text=example` to reverse the text.

**Business Logic:** Validate that the text parameter is not empty.

**Outcome:** Understand RESTful principles.

## Practice 6

# HTTP Methods (GET, POST, PUT, DELETE)

## Exercise: CRUD API

Build a RESTful API for managing a ToDo list with GET, POST, PUT, and DELETE methods.

**Business Logic:** Add validation for POST and PUT to ensure the title field is not empty.

**Outcome:** Learn how to implement CRUD operations.

## Exercise: Resource Validation

Create a Books API with:

GET `/books`: Returns all books.

POST `/books`: Adds a new book.

**Business Logic:** Return a 409 Conflict if a book with the same title already exists.

**Outcome:** Practice HTTP method implementation.

## Exercise: HTTP Method Error Handling

Extend the ToDo API to handle cases where:

A PUT request tries to update a non-existent task.

A DELETE request tries to delete a non-existent task.

**Business Logic:** Return 404 Not Found with meaningful error messages.

**Outcome:** Improve error handling in REST APIs.

## Practice 7

## Exercise: First API with Swagger

Set up a new .NET Core Web API project in Visual Studio/VS Code.

**Business Logic:** Add Swagger for API documentation and test a simple "Ping" endpoint.

**Outcome:** Learn the setup process and Swagger integration.

## Exercise: Debugging

Create a simple Web API with one endpoint and set breakpoints to debug it.

**Business Logic:** Add logic to calculate the square of a number and debug step-by-step.

**Outcome:** Learn debugging basics.

#### Exercise: File Structure

Create a Web API project and reorganize the default folder structure:

Move models, services, and controllers into their respective folders.

**Business Logic:** Add a simple service that returns a hardcoded message.

**Outcome:** Understand the structure of a .NET Core project.

#### Practice 8

##### Exercise: Dynamic Hello World

Create a Web API with an endpoint `/hello` that accepts a query parameter name and returns "Hello, [name]!".

**Business Logic:** If no name is provided, default to "World".

**Outcome:** Practice dynamic responses.

##### Exercise: Logging in Hello World

Add logging to the "Hello World" API to log every request and its parameters.

**Business Logic:** Log the current time and user agent for each request.

**Outcome:** Learn basic logging.

##### Exercise: API Versioning

Add versioning to the "Hello World" API (`/v1/hello` and `/v2/hello`).

**Business Logic:** In version 2, return the greeting in uppercase.

**Outcome:** Practice API versioning.

### Assignment Exercise

#### Assignment 1

## Assignment: Build a Mini Student Management System with .NET Core Web API

### Objective

To combine knowledge of .NET Core Web API, RESTful principles, HTTP methods, and basic features of .NET Core into a practical assignment.

### Problem Statement

Develop a Student Management System API using .NET Core Web API that allows users to perform CRUD operations on student data. The system should provide

endpoints for managing student information such as name, age, and grade. Additionally, implement error handling, validations, and logging.

## Requirements

### Part 1: Project Setup

Set up a new .NET Core Web API project in **Visual Studio** or **VS Code**.

Organize the project into folders for Models, Controllers, and Services.

### Part 2: Features and Endpoints

#### Get All Students

**Endpoint:** GET /students

**Description:** Returns a list of all students.

**Business Logic:** Return an appropriate message if no students exist.

#### Get Student by ID

**Endpoint:** GET /students/{id}

**Description:** Fetches the details of a specific student by their ID.

**Business Logic:** If the ID does not exist, return a 404 Not Found with an error message.

#### Add a New Student

**Endpoint:** POST /students

**Description:** Adds a new student to the system.

**Business Logic:**

Validate that the name is not empty, age is between 5 and 100, and grade is a valid letter (A–F).

Return a 400 Bad Request for invalid inputs.

#### Update Student Details

**Endpoint:** PUT /students/{id}

**Description:** Updates the details of an existing student.

**Business Logic:**

Check if the student exists before updating.

Only update fields that are provided in the request.

#### Delete a Student

**Endpoint:** DELETE /students/{id}

**Description:** Deletes a student by their ID.

**Business Logic:**

Prevent deletion if the student is in grade A.

Return a 400 Bad Request with a message like "Cannot delete top-performing students".

## Part 3: Additional Requirements

### Logging

Implement logging for every API call with details like the endpoint, HTTP method, and timestamp.

### Configuration

Use appsettings.json to store application settings such as the maximum number of students allowed (e.g., 100).

### Error Handling

Implement global error handling using middleware to catch unhandled exceptions and return meaningful error messages.

### Asynchronous Programming

Use async/await for all database or data layer operations.

### Expected Output

A functional RESTful API that meets all requirements.

Well-structured code with proper error handling and logging.

Validation messages and meaningful error responses for invalid inputs.

### Evaluation Criteria

**Code Quality:** Clean and modular code with proper folder structure.

**Functionality:** All endpoints work as expected with valid and invalid inputs.

**RESTful Principles:** Proper use of HTTP methods and status codes.

**Error Handling and Logging:** Clear error messages and logged API activities.

**Completion:** Ability to complete the assignment within the time limit.

Online Reference

No online Reference

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 14

## What to learn

### Controllers and Routing

Understanding Controllers in Web API

Attribute-based routing

Conventional routing

Parameter binding in routes

Hands-on: Create an API for managing "Books" with CRUD endpoints.

## Practice Exercise

### Practice 1

#### Create a Book Controller with Attribute-Based Routing

**Objective:** Create a controller for managing books using **attribute-based routing**.

**Task:**

Implement GET, POST, PUT, and DELETE methods with appropriate attribute-based routes ([Route] and [Http\*] attributes).  
Use routes like /api/books and /api/books/{id}.

### Practice 2

#### Implement CRUD Operations Using Conventional Routing

**Objective:** Set up **conventional routing** for a BookController.

**Task:**

Configure the route in Program.cs as  
api/{controller}/{action}/{id?}.  
Implement methods such as GetBooks, GetBookById, AddBook, UpdateBook, and DeleteBook.

### Practice 3

#### Fetch Books by Author Using Parameter Binding

**Objective:** Fetch books by a specific author using **parameter binding** in the route.

**Task:**

Create an endpoint like /api/books/author/{authorName}.  
Return a list of books written by the specified author. If no books are found, return an appropriate message.

### Practice 4



## Implement Pagination for Books

**Objective:** Implement a GET endpoint for paginated results.

**Task:**

Add an endpoint like `/api/books?pageNumber={pageNumber}&pageSize={pageSize}`.

Business Logic: Return the specified page of books, or an error message if the requested page exceeds the available data.

### Practice 5

## Update a Book's Price with Validation

**Objective:** Implement a PUT endpoint to update a book's price.

**Task:**

Endpoint: `/api/books/{id}/update-price`

Business Logic: Ensure the new price is greater than zero. Return 400 Bad Request if invalid.

### Practice 6

## Delete a Book with Conditional Logic

**Objective:** Delete a book by ID with additional validation.

**Task:**

Prevent deletion if the book has more than 100 copies in stock. Return an appropriate message if deletion is not allowed.

### Practice 7

## Retrieve Books Published in a Specific Year

**Objective:** Create a GET endpoint to fetch books published in a specific year.

**Task:**

Route: `/api/books/year/{year}`.

Business Logic: If no books are found for the specified year, return a 404 Not Found.

### Practice 8

## Search Books by Title Using Query Parameters

**Objective:** Implement a GET endpoint to search for books by title.

**Task:**

Route: `/api/books/search?title={title}`.

Business Logic: Perform a case-insensitive search. If no matches are found, return a helpful error message.

### Practice 9

## Add a New Book with Custom Validation

**Objective:** Implement a POST endpoint to add a new book.

**Task:**

Business Logic: Validate that the title is not empty, the author name has at least 3 characters, and the price is greater than 0. Return a 400 Bad Request if any validation fails.

## Practice 10

### Implement Sorting for Books

**Objective:** Create an endpoint to fetch all books sorted by title or price.

**Task:**

Route: `/api/books?sortBy={title|price}`.

Business Logic: Sort the books in ascending or descending order based on a query parameter like `order=asc|desc`.

## Assignment Exercise

### Assignment 1

Create a Restful API to create an employee, get all employees, get an employee, get an employee, update and employee `http://localhost:3000/emps` •  
AddressLine1(optional): string • AddressLine2(optional): string •  
AddressLine3(optional): string • assignments(optional): array •  
CitizenshipId(optional): integer(int64) • CitizenshipLegislationCode(optional): string •  
CitizenshipStatus(optional): string • CitizenshipToDate(optional): string(date) •  
City(optional): string • CorrespondenceLanguage(optional): string •  
Country(optional): string • CreationDate(optional): string(date-time) •  
DateOfBirth(optional): string(date) • directReports(optional): array •  
DisplayName(optional): string • DriversLicenseExpirationDate(optional): string(date) •  
DriversLicenseId(optional): integer(int64) •  
DriversLicenseIssuingCountry(optional): string • EffectiveStartDate(optional): string(date) •  
Ethnicity(optional): string • FirstName(optional): string •  
Gender(optional): string • HireDate(optional): string(date) •  
HomeFaxAreaCode(optional): string • HomeFaxCountryCode(optional): string •  
HomeFaxExtension(optional): string • HomeFaxLegislationCode(optional): string •  
HomeFaxNumber(optional): string • HomePhoneAreaCode(optional): string •  
HomePhoneCountryCode(optional): string • HomePhoneExtension(optional): string •  
HomePhoneLegislationCode(optional): string • HomePhoneNumber(optional): string •  
Honors(optional): string • LastName(optional): string •  
LastUpdateDate(optional): string(date-time) • LegalEntityId(optional): integer(int64) •  
LicenseNumber(optional): string • links(optional): array •  
MaritalStatus(optional): string • MiddleName(optional): string •  
MilitaryVetStatus(optional): string • NameSuffix(optional): string •  
NationalId(optional): string • NationalIdCountry(optional): string Assignments:  
Assignment Fields • ActionCode(optional): string • ActionReasonCode(optional):

string • ActualTerminationDate(optional): string(date) •  
AssignmentCategory(optional): string • assignmentDFF(optional): array •  
assignmentExtraInformation(optional): array • AssignmentId(optional):  
integer(int64) • AssignmentName(optional): string • AssignmentNumber(optional):  
string • AssignmentProjectedEndDate(optional): string(date) •  
AssignmentStatus(optional): string • AssignmentStatusTypeId(optional):  
integer(int64) • BusinessUnitId(optional): integer(int64) • CreationDate(optional):  
string(date-time) • DefaultExpenseAccount(optional): string •  
DepartmentId(optional): integer(int64) • EffectiveEndDate(optional): string(date) •  
EffectiveStartDate(optional): string(date) • empreps(optional): array •  
EndTime(optional): string • Frequency(optional): string • FullPartTime(optional):  
string • GradeId(optional): integer(int64) • GradeLadderId(optional): integer(int64)  
• JobId(optional): integer(int64) • LastUpdateDate(optional): string(date-time) •  
LegalEntityId(optional): integer(int64) • links(optional): array • LocationId(optional):  
integer(int64) • ManagerAssignmentId(optional): integer(int64) •  
ManagerId(optional): integer(int64) Create an Assignments API  
http://localhost:3000/emps/{empID}/child/assignments Get All Assignments  
http://localhost:3000/emps/{empID}/child/assignments Get an Assignment  
http://localhost:3000/emps/{empID}/child/assignments/{AssignmentID} Update  
an assignment  
http://localhost:3000/emps/{empID}/child/assignments/{AssignmentID}

### Assignment 2

## Assignment: Build a Complete "Library Management" Web API

### Objective

Develop a Web API for managing a library system. The API should allow users to perform CRUD operations on books, implement routing strategies, and validate parameters. The assignment should demonstrate understanding of controllers, routing (attribute-based and conventional), parameter binding, and RESTful practices.

### Assignment Requirements

You need to create a Library Management API that fulfills the following requirements:

#### 1. Create the Project

**Task:** Create a new .NET Core Web API project named LibraryManagement.  
**Tools:** Use Visual Studio or VS Code.

Configure the development environment and set up the project.

## 2. Implement the Book Model

Properties:

- Id (integer, auto-increment)
- Title (string, required)
- Author (string, required)
- Price (decimal, must be greater than 0)
- YearPublished (integer, must be a valid year)
- Stock (integer, default value 0)

## 3. Create the Controller

Create a BookController with the following endpoints:

### 3.1. Attribute-Based Routing for CRUD

GET /api/books: Retrieve all books.

GET /api/books/{id}: Retrieve a book by its ID.

**Validation:** Return a 404 Not Found if the book does not exist.

POST /api/books: Add a new book.

**Validation:** Ensure Title, Author, Price, and YearPublished are valid.

Return 400 Bad Request for invalid data.

PUT /api/books/{id}: Update a book's details by ID.

**Validation:** Ensure the book exists before updating.

DELETE /api/books/{id}: Delete a book by ID.

**Business Logic:** Prevent deletion if Stock > 50 and return an appropriate message.

### 3.2. Custom Routes

GET /api/books/author/{authorName}: Retrieve all books by a specific author.

**Validation:** Return a helpful message if no books are found.

GET /api/books/year/{year}: Retrieve books published in a specific year.

## 4. Conventional Routing

Configure conventional routing in Program.cs to handle requests as:

bash

Copy code

```
api/{controller}/{action}/{id?}
```

Ensure actions in the controller match the route structure:

Example: /api/books/GetByTitle?title={title} should map to GetByTitle().

## 5. Business Logic

Implement the following:

**Pagination:** Add a GET `/api/books?pageNumber={pageNumber}&pageSize={pageSize}` endpoint. Return paginated results.

**Sorting:** Allow books to be sorted by Title or Price using `/api/books?sortBy=title|price&order=asc|desc`.

**Stock Validation:** Only return books with Stock > 0 in `/api/books`.

## 6. Error Handling

Handle common errors (e.g., 404 Not Found, 400 Bad Request) with meaningful messages.

Ensure validation messages are clear and descriptive.

## 7. Testing

**Manual Testing:**

Use tools like Postman or Swagger to test all endpoints.

**Automated Testing:**

Write unit tests for key endpoints like `AddBook`, `GetBooksByAuthor`, and `UpdateBook`.

## Deliverables

The completed .NET Core Web API project.

A README.md file with:

Instructions on how to run the project.

Sample requests for each endpoint.

Details of the validation rules.

Postman collection (optional).

Online Reference

No online Reference

.NET Core Web API

WEB API (old)

Authentication And Authorization (WEBAPI)(old)

FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 15

## What to learn

### Models and Data Transfer

- Creating models in Web API

- Data annotations and validations

- Model binding and validation

- Custom Validation in .NET Core

- Hands-on: Enhance the "Books" API with model validation.

- Hands-on Custom Validation

## Practice Exercise

### Practice 1

Repository Pattern

### Practice 2

## Creating Models in Web API

Design a model class named **Book** with properties: **Id** (int), **Title** (string), **Author** (string), **PublishedYear** (int), and **Genre** (string). Ensure that the **Id** is an auto-incrementing primary key.

### Practice 3

## Data Annotations and Validations

Enhance the **Book** model by adding data annotations for validation. For example, make the **Title** property required and ensure that **PublishedYear** is a valid year (greater than 0).

### Practice 4

## Model Binding and Validation

Implement model binding in a controller action that accepts a **Book** object. Ensure that the model is validated and return appropriate error messages if validation fails.

### Practice 5

## Enhancing the 'Books' API with Model Validation

Modify the existing **BooksController** to include model validation. If a **Book** object fails validation, return a 400 Bad Request response with the validation errors.

## Practice 6

# Creating a Custom Validation Attribute

Create a custom validation attribute named **ValidGenreAttribute** that checks if the **Genre** property of the **Book** model is one of the predefined genres (e.g., Fiction, Non-Fiction, Science, History). Apply this attribute to the **Genre** property.

## Assignment Exercise

### Assignment 1

## Assignment: Building a Comprehensive 'Books' API

**Functional Flow:** Create a RESTful API for managing a collection of books. The API should support the following operations:

- GET all books
- GET a book by ID
- POST a new book
- PUT to update an existing book
- DELETE a book

Ensure that all operations validate the **Book** model using data annotations and return appropriate HTTP status codes.

**Business Logic:** Implement the following rules:

- The **Title** must be unique.
- The **PublishedYear** must be a valid year (greater than 0).
- The **Genre** must be one of the predefined genres.

Use in-memory storage for simplicity, and ensure that all outputs are displayed on the user interface (UI) only, not in the console.

## Online Reference

No online Reference

## .NET Core Web API

## WEB API (old)

## Authentication And Authorization (WEBAPI)(old)

## FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 16

## What to learn

### Dependency Injection (DI)

What is Dependency Injection?

Built-in DI in .NET Core

Configuring services

Repository Pattern and its implementation

Hands-on Generic Repository Pattern, register and consume repository

Hands-on: Creating service which inherit from generic repository and injecting the controller.

## Practice Exercise

### Practice 1

do the hands on the following topic from the videos

### Practice 2

## Understanding Dependency Injection

Explain what Dependency Injection (DI) is and how it is used in .NET Core applications. Provide a simple code example demonstrating the concept of DI in a .NET Core Web API project.

### Practice 3

## Implementing Built-in DI in .NET Core

Using the built-in Dependency Injection framework in .NET Core, create a service that fetches book details from a static list. Implement this service in a controller and demonstrate how to inject it using DI.

### Practice 4

## Configuring Services in Startup.cs

In a new .NET Core Web API project, configure a custom service in the Startup.cs file. Explain the purpose of the ConfigureServices method and how to register a service for DI.

### Practice 5

## Authentication and Authorization Basics

Define the concepts of Authentication and Authorization in the context of a Web API. Provide a code snippet that demonstrates how to set up basic authentication in a .NET Core Web API application.

### Practice 6

## Using JSON Web Token (JWT)

Explain what JSON Web Tokens (JWT) are and how they are used for securing APIs. Provide an example of generating a JWT token upon user login and how to validate it in a .NET Core Web API.

### Practice 7

## Adding a Service Layer to the 'Books' API

Enhance an existing 'Books' API by adding a service layer. Create a service that handles business logic related to book management, such as adding, updating, and retrieving book information.

### Practice 8

## Hands-on Authentication and Authorization

Implement a simple authentication and authorization mechanism in a .NET Core Web API. Create endpoints that require user roles and demonstrate how to restrict access based on user roles.

## Assignment Exercise

### Assignment 1



Create A Database design and insert record in the database. Create views In a hospital there are different departments. Patients are treated in these departments by the doctors assigned to patients. Usually each patient is treated by a single doctor, but in rare cases they will have two or three. Healthcare assistants will also attend to patients; every department has many healthcare assistants. Each patient is required to take a variety of drugs during different parts of the day such as morning, afternoon and night. Login Module API After login only below module should be accessible. If he is admin can update these details and others using only view the output. After Creating a Database design. Create ORM with database first Approach. Insert a Doctor Update a Doctor Delete a Doctor Find a report of patient assigned to a particular doctor Find a report of medicine list for a particular patient Display summary report of Doctor and patient (use Include method) After successfully login only these API should be accessible.

Assignment 2

Building a Comprehensive 'Books' API with DI, Authentication, and Authorization

Develop a complete 'Books' API that incorporates Dependency Injection, Authentication, and Authorization. The API should allow users to perform CRUD operations on books, but only authenticated users should be able to add or modify books. Use JWT for securing the API endpoints. The functional flow should include:

- User registration endpoint that stores user credentials securely.
- User login endpoint that generates a JWT token upon successful authentication.
- Endpoints for managing books (GET, POST, PUT, DELETE) that check for user authentication and roles.
- Ensure that the API returns appropriate HTTP status codes and messages based on the success or failure of operations.

Business Logic:

- Only users with the 'Admin' role can add or delete books.
- All authenticated users can view the list of books.
- Implement error handling for unauthorized access attempts.

Ensure that all outputs are displayed on the user interface (UI) and not in the console. Provide a detailed README file explaining how to set up and test the API.

Assignment 3 (68257f9be12dd826bd9eb7f5)

API Endpoints for Bill Generation and Payment System

1. User Registration (With Role Assignment)

/api/v1/users/register

Method Payload

POST	{ "email": "user@example.com", "password": "securePass123", "username": "user123", "role": "customer" } or { "email": "admin@example.com", "password": "securePass123", "username": "adminUser", "role": "admin" }
------	--

Property Name Validation Error Message			Status Code
email	Required	Email is required	400
	Format	Email is invalid	400
	Exists	User with this email already exists	400
username	Required	Username is required	400
	Length	Username must be at least 3 characters long	400
	Exists	User with this username already exists	400
password	Required	Password is required	400
role	Required	Role is required	400
	--	Role must be either "customer" or "admin"	400

Responses	Status Code	Body
Success	201	{ "data": { "message": "User registered successfully", "userId": "unique_id_here" } }
Validation Error	400	{ "errors": [ { "field": "email", "message": "Email is invalid" }, { "field": "role", "message": "Role is required" } ] }
Conflict	400	{ "message": "User with this email or username already exists" }

2. User Login

/api/v1/users/login

Method Payload

POST	{ "username": "user123", "password": "securePass123" }
------	--

Property Name	Validation	Error Message	Status Code
username	Required	Username is required	400
password	Required	Password is required	400

Responses	Status Code	Body
Success	200	{ "data": { "userId": "unique_id_here", "accessToken": "Bearer Token", "role": "customer" } }
Invalid Credentials	401	{ "message": "Invalid user credentials" }

### 3. Generate Bill (Admin Only)

#### /api/v1/bills/generate

##### Method Payload

POST { "customerId": "customer\_id\_here", "amount": 100, "dueDate": "YYYY-MM-DD" }

Property Name	Validation	Error Message	Status Code
customerId	Required	Customer ID is required	400
amount	Required	Amount is required	400
	Positive	Amount must be a positive number	400
dueDate	Required	Due date is required	400
	Date Format	Due date must be in YYYY-MM-DD format	400

Responses	Status Code	Body
Success	201	{ "data": { "message": "Bill generated successfully", "billId": "unique_bill_id" } }
Validation Error	400	{ "errors": [ { "field": "amount", "message": "Amount must be a positive number" } ] }
Unauthorized	403	{ "message": "Unauthorized access. Admin privileges are required." }

### 4. Admin View All Bills

#### /api/v1/bills

##### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "data": [ { "billId": "unique_bill_id", "customerId": "customer_id_here", "amount": 100, "status": "Unpaid", "dueDate": "YYYY-MM-DD" }, ... ] }
No Bills	404	{ "message": "No bills found" }

### 5. Get Customer Bills (View For customers)

#### /api/v1/bills/customer

##### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "data": [ { "billId": "unique_bill_id", "amount": 100, "status": "Unpaid", "dueDate": "YYYY-MM-DD" }, ... ] }
No Bills	404	{ "message": "No bills found for this customer" }

### 6. Pay Bill

#### /api/v1/bills/pay/{billId}

##### Method Payload

PATCH { "status": "Paid" }

Property Name	Validation	Error Message	Status Code
status	Required	Bill status is required	400
	--	Status must be "Paid"	400

Responses	Status Code	Body
Success	200	{ "data": { "message": "Bill payment successful", "billId": "unique_bill_id", "newStatus": "Paid" } }

Responses	Status Code	Body
Bill Not Found	404	{ "message": "Bill not found" }
Already Paid	400	{ "message": "Bill has already been paid" }
Unauthorized	403	{ "message": "Unauthorized access. Only the customer can pay the bill." }

## Task Objective

Build a RESTful API for a Bill Generation and Payment System using JavaScript with any backend framework. The API should allow:

- User registration with a specified role ("customer" or "admin") for users.
- User login functionality for both customers and admin users.
- Admin users should be able to generate bills for specific customers and view all bills along with their status (default is "Unpaid").
- Customers should be able to view their bills and make payments for their respective bills by providing the bill ID and changing the status to "Paid".

Ensure proper validation, meaningful error messages, and include scenarios that result in specific HTTP status codes to uphold business logic.

Assignment 4 (6826e2b924c5af26cafd7eec)

## API Endpoints for Enhanced Country Information Application

### Objective

Build a RESTful API that enables a mobile or web application to display a list of countries, view detailed information, manage user preferences for favorite countries, and implement additional business logic pieces to enhance the functionality of the application. The API should be thoroughly designed, adhere to best practices, and incorporate user-related functionalities.

### Note on Implementation

Always delete or reset the database before running test cases, as required by the question.

### Sample Data for Country Details

Detail of Countries:

```
{
  "countries": {
    "FR": {
      "countryCode": "FR",
      "name": "France",
      "capital": "Paris",
      "population": 65273511,
      "region": "Europe",
      "languages": ["French"],
      "currency": { "code": "EUR", "name": "Euro" },
      "area": 551695,
      "demonym": "French",
      "timezone": "UTC+1",
      "rating": 4.5
    },
    "DE": {
      "countryCode": "DE",
      "name": "Germany",
      "capital": "Berlin",
      "population": 83783942,
      "region": "Europe",
      "languages": ["German"],
      "currency": { "code": "EUR", "name": "Euro" },
      "area": 357022,
      "demonym": "German",
      "timezone": "UTC+1",
      "rating": 4.0
    },
    "JP": {
      "countryCode": "JP",
      "name": "Japan",
      "capital": "Tokyo",
      "population": 126476461,
      "region": "Asia",
      "languages": ["Japanese"],
      "currency": { "code": "JPY", "name": "Yen" },
      "area": 377975,
      "demonym": "Japanese",
      "timezone": "UTC+9",
      "rating": 4.8
    },
    "BR": {
      "countryCode": "BR",
      "name": "Brazil",
      "capital": "Brasília",
      "population": 212559417,
      "region": "South America",
      "languages": ["Portuguese"],
      "currency": { "code": "BRL", "name": "Real" },
      "area": 8515767,
      "demonym": "Brazilian",
      "timezone": "UTC-3",
      "rating": 4.6
    },
    "AU": {
      "countryCode": "AU",
      "name": "Australia",
      "capital": "Canberra",
      "population": 25499884,
      "region": "Oceania",
      "languages": ["English"],
      "currency": { "code": "AUD", "name": "Australian Dollar" },
      "area": 7692024,
      "demonym": "Australian",
      "timezone": "UTC+10",
      "rating": 4.3
    },
    "CA": {
      "countryCode": "CA",
      "name": "Canada",
      "capital": "Ottawa",
      "population": 37742154,
      "region": "North America",
      "languages": ["English", "French"],
      "currency": { "code": "CAD", "name": "Canadian Dollar" },
      "area": 9984670,
      "demonym": "Canadian",
      "timezone": "UTC-5",
      "rating": 4.2
    },
    "IN": {
      "countryCode": "IN",
      "name": "India",
      "capital": "New Delhi",
      "population": 1380004385,
      "region": "Asia",
      "languages": ["Hindi", "English"],
      "currency": { "code": "INR", "name": "Indian Rupee" },
      "area": 3287263,
      "demonym": "Indian",
      "timezone": "UTC+5:30",
      "rating": 4.1
    },
    "IT": {
      "countryCode": "IT",
      "name": "Italy",
      "capital": "Rome",
      "population": 60244639,
      "region": "Europe",
      "languages": ["Italian"],
      "currency": { "code": "EUR", "name": "Euro" },
      "area": 301340,
      "demonym": "Italian",
      "timezone": "UTC+1",
      "rating": 4.5
    },
    "ES": {
      "countryCode": "ES",
      "name": "Spain",
      "capital": "Madrid",
      "population": 46754778,
      "region": "Europe",
      "languages": ["Spanish"],
      "currency": { "code": "EUR", "name": "Euro" },
      "area": 505992,
      "demonym": "Spanish",
      "timezone": "UTC+1",
      "rating": 4.4
    },
    "US": {
      "countryCode": "US",
      "name": "United States",
      "capital": "Washington, D.C.",
      "population": 331002651,
      "region": "North America",
      "languages": ["English"],
      "currency": { "code": "USD", "name": "United States Dollar" },
      "area": 9372610,
      "demonym": "American",
      "timezone": "UTC-5",
      "rating": 4.6
    },
    "RU": {
      "countryCode": "RU",
      "name": "Russia",
      "capital": "Moscow",
      "population": 145912025,
      "region": "Europe/Asia",
      "languages": ["Russian"],
      "currency": { "code": "RUB", "name": "Russian Ruble" },
      "area": 17098242,
      "demonym": "Russian",
      "timezone": "UTC+3",
      "rating": 3.9
    },
    "ZA": {
      "countryCode": "ZA",
      "name": "South Africa",
      "capital": "Pretoria",
      "population": 59308690,
      "region": "Africa",
      "languages": ["Afrikaans", "English", "Zulu"],
      "currency": { "code": "ZAR", "name": "South African Rand" },
      "area": 1219090,
      "demonym": "South African",
      "timezone": "UTC+2",
      "rating": 4.1
    },
    "MX": {
      "countryCode": "MX",
      "name": "Mexico",
      "capital": "Mexico City",
      "population": 128635261,
      "region": "North America",
      "languages": ["Spanish"],
      "currency": { "code": "MXN", "name": "Mexican Peso" },
      "area": 1964375,
      "demonym": "Mexican",
      "timezone": "UTC-6",
      "rating": 4.2
    }
  }
}
```

"Mexico", "capital": "Mexico City", "population": 128932753, "region": "North America", "languages": ["Spanish"], "currency": { "code": "MXN", "name": "Mexican Peso" }, "area": 1964375, "demonym": "Mexican", "timezone": "UTC-6", "rating": 4.2 }, "GB": { "countryCode": "GB", "name": "United Kingdom", "capital": "London", "population": 67886011, "region": "Europe", "languages": ["English"], "currency": { "code": "GBP", "name": "British Pound" }, "area": 243610, "demonym": "British", "timezone": "UTC+0", "rating": 4.3 }, "MY": { "countryCode": "MY", "name": "Malaysia", "capital": "Kuala Lumpur", "population": 32365999, "region": "Asia", "languages": ["Malay", "English"], "currency": { "code": "MYR", "name": "Malaysian Ringgit" }, "area": 330803, "demonym": "Malaysian", "timezone": "UTC+8", "rating": 4.0 }, "ID": { "countryCode": "ID", "name": "Indonesia", "capital": "Jakarta", "population": 273523615, "region": "Asia", "languages": ["Indonesian"], "currency": { "code": "IDR", "name": "Indonesian Rupiah" }, "area": 1904569, "demonym": "Indonesian", "timezone": "UTC+7", "rating": 4.1 } }

## Enhanced Business Logic Requirements

- Each user should be able to save multiple preferred countries.
- Users should be able to rate their preferred countries (1 to 5 stars).
- The API should allow users to get a list of countries based on their ratings.
- Implement search functionality to find countries by name or capital.

### 1. Get List of Countries

#### /api/v1/countries

##### Method Payload

GET N/A

##### Responses Status Code Body

Success	200	{ "data": [...], "message": "Countries fetched successfully", "success": true }
---------	-----	---

### 2. Search Countries

#### /api/v1/countries/search

##### Method Payload

GET { "query": "France" }

##### Property Name Validation Error Message Status Code

query	Required	Search query is required	400
-------	----------	--------------------------	-----

Responses	Status Code	Body
Success	200	{ "data": <Country-specific data: France>, "message": "Countries matching the query fetched successfully." }
Not Found	404	{ "message": "No countries found matching your query." }

### 3. Get Details of a Selected Country

#### /api/v1/countries/{countryCode}

##### Method Payload

GET N/A

Responses	Status Code	Body
Success	200	{ "data": <Country-specific data: countryCode>, "message": "Country details fetched successfully", "success": true }
Not Found	404	{ "message": "Country not found" }

### 4. Save Preferred Country with Rating

#### /api/v1/preferences/countries

##### Method Payload

POST { "countryCode": "FR", "rating": 5 }

##### Property Name Validation Error Message Status Code

countryCode	Required	Country code is required	400
-------------	----------	--------------------------	-----

Property Name	Validation Error Message	Status Code
rating	Required Rating is required	400
	Range Rating must be between 1 and 5	400
	Exists Country must exist in the database	404

Responses	Status Code	Body
Success	201	{ "message": "Country saved as preferred with rating successfully" }
Validation Error	400	{ "message": "Rating must be between 1 and 5" }
Not Found	404	{ "message": "Country not found" }

## 5. Remove Preferred Country

/api/v1/preferences/countries/{countryCode}

Method Payload

DELETE N/A

Property Name	Validation Error Message	Status Code
countryCode	Required Country code is required	400
	Exists Country must exist in preferences	404

Responses	Status Code	Body
Success	200	{ "message": "Country removed from preferred successfully" }
Validation Error	400	{ "message": "Country code is required" }
Not Found	404	{ "message": "Country must exist in preferences" }

## 6. Get Preferred Countries with Ratings

/api/v1/preferences/countries

Method Payload

GET N/A

Responses	Status Code	Body
Success	200	{ "data": [ <countries data>... ], "message": "Preferred countries fetched successfully" }
Not Found	404	{ "message": "No preferred countries found" }

Assignment 5 (68283007e12dd826bd9eb83d)

## API Endpoints for Job Application Portal with Admin User Implementation

### Task Objective

Build an API for a Job Application Portal that includes user registration with a role assignment (either 'user' or 'admin'), login functionality, and manages job postings by admin users. Admin users will have special privileges to manage job applications, while normal users will only be able to apply for jobs and view their applications. Ensure proper validation and meaningful error messages are provided in the API responses, including scenarios that result in 409 Conflict status codes to handle specific business logic conflicts.

### 1. User Registration (With Role Assignment)

/api/v1/users/register

Method Payload

POST { "email": "user@example.com", "password": "securePass123", "username": "applicantUser", "role": "user" } or { "email": "admin@example.com", "password": "securePass123", "username": "adminUser", "role": "admin" }

Property Name	Validation Error Message	Status Code
email	Required Email is required	400
	Format Email is invalid	400
	Exists User with this email already exists	409

Property Name	Validation Error Message	Status Code
username	Required	Username is required 400
	Length	Username must be at least 3 characters long 400
	Exists	User with this username already exists 409
password	Required	Password is required 400
role	Required	Role is required 400
	--	Role must be one of the following: ["user", "admin"] 400

Responses	Status Code	Body
Success	201	{ "data": { "message": "User registered successfully", "userId": "unique_id_here" } }
Validation Error	400	{ "errors": [ { "field": "email", "message": "Email is invalid" }, { "field": "role", "message": "Role must be either 'user' or 'admin'" } ] }
Conflict	409	{ "message": "User with this email or username already exists" }

## 2. User Login

### /api/v1/users/login

#### Method Payload

POST { "username": "applicantUser", "password": "securePass123" }

Property Name	Validation Error Message	Status Code
username	Required	Username is required 400
password	Required	Password is required 400

Responses	Status Code	Body
Success	200	{ "data": { "userId": "unique_id_here", "accessToken": "Bearer Token", "role": "user" } }
Invalid Credentials	401	{ "message": "Invalid user credentials" }

## 3. Job Posting (Admin Only)

### /api/v1/jobs

#### Method Payload

POST { "title": "Software Engineer", "description": "Job description goes here", "position": "Software Developer" }

Property Name	Validation Error Message	Status Code
title	Required	Job title is required 400
description	Required	Job description is required 400
position	Required	Position is required 400

Responses	Status Code	Body
Success	201	{ "data": { "message": "Job posted successfully", "jobId": "unique_id_here" } }
Validation Error	400	{ "errors": [ { "field": "title", "message": "Job title is required" } ] }
Unauthorized	401	{ "message": "Unauthorized access. Admin privileges are required." }

## 4. User Job Application

### /api/v1/job-applications

#### Method Payload

POST { "email": "applicant@gmail.com", "name": "John Doe", "positionId": "jobId\_here" }

Property Name	Validation Error Message	Status Code
email	Required	Email is required 400
	Format	Email is invalid 400
	Exists	An application with this email already exists 409
name	Required	Name is required 400
positionId	Required	Position ID is required 400
	Exists	Provided job ID does not exist 404

Responses	Status Code	Body
Success	201	{ "data": { "message": "Job application submitted successfully", "applicationId": "unique_id_here" } }
Validation Error	400	{ "errors": [ { "field": "email", "message": "Email is invalid" } ] }
Conflict	409	{ "message": "An application with this email already exists" }
Not Found	404	{ "message": "Provided job ID does not exist" }

## 5. Admin: View All Job Applications

/api/v1/jobs/{jobId}/applications

### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "data": [ { "applicationId": "id_here", "email": "applicant@gmail.com", "name": "John Doe", "status": "Under Review" }, ... ] }
Not Found	404	{ "message": "No applications found for this job" }
Unauthorized	403	{ "message": "Unauthorized access. Admin privileges are required." }

## 6. View User's Own Applications

/api/v1/my-applications

### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "data": [ { "applicationId": "id_here", "email": "applicant@gmail.com", "name": "John Doe", "status": "Under Review" }, ... ] }
Not Found	404	{ "message": "No applications found for this user" }

## 7. Admin: Update Job Application Status

/api/v1/job-applications/{applicationId}/status

### Method Payload

PATCH { "status": "Interview Scheduled" }

Property Name	Validation Error Message	Status Code
status	Required Status is required	400
--	Status must be one of the following: ["Under Review", "Interview Scheduled", "Rejected", "Hired"]	400
Exists	Application status cannot be updated if already marked as 'Rejected' or 'Hired'	409

Responses	Status Code	Body
Success	200	{ "data": { "message": "Application status updated successfully", "applicationId": "unique_id_here", "newStatus": "Interview Scheduled" } }
Validation Error	400	{ "errors": [ { "field": "status", "message": "Status must be one of the following: ['Under Review', 'Interview Scheduled', 'Rejected', 'Hired']" } ] }
Conflict	409	{ "message": "Application status cannot be updated if already marked as 'Rejected' or 'Hired'" }
Not Found	404	{ "message": "Application not found" }
Unauthorized	403	{ "message": "Unauthorized access. Admin privileges are required." }

Assignment 6 (682da8195611ba4b46206687)

API Endpoints for Vehicle Service Booking Management System (VSBMS)

You are required to build a RESTful API for a **Vehicle Service Booking Management System (VSBMS)** in .NET Core Web API. The system includes user management, vehicle CRUD operations, and booking services. It also incorporates features like JWT-based authentication and role-based authorization, custom validation, middleware handling, file uploads, soft deletes, and automatic audit tracking.

## Special Instructions

### Entity Metadata

All entities must include the following fields:

CreatedById  
ModifiedById  
CreatedDate  
ModifiedDate


### Soft Delete

All delete operations should implement **soft delete** using an IsDeleted flag. Soft-deleted records must not appear in API results.

### Image Upload (Vehicle)

Image should be uploaded with the vehicle POST request using multipart/form-data.  
Save image to the Uploads folder.  
Register Uploads as a static file folder.  
Store the relative URL (e.g., /Uploads/car.png) in the database.

### Date-Sensitive Testing Note

 The current date is 19 May 2025. Ensure booking date validation handles future dates dynamically so automated tests behave correctly.

## Authentication & Authorization

Use JWT Bearer Tokens  
Roles: admin, customer  
Use role-based authorization on endpoints

403 Forbidden Response Example:

```
{ "message": "You are not authorized to perform this action." }
```

## Business Rules (Service Charges)

Booking charges must be automatically applied based on the selected serviceType:

### Service Type Charge (INR)

Basic	500
Premium	1000
Full	1500

 The charge is not passed by the client. It is calculated server-side and stored with the booking.

## 1. User Registration

/api/v1/auth/register

### Method Payload

POST { "name": "John Doe", "email": "john@example.com", "phone": "9999999999", "password": "Pass@123", "role": "customer" }

Property Name Validation		Error Message	Status Code
name	Required	"Name is required."	400
email	Required, Email, Unique	"A valid and unique email is required."	400
phone	Required, 10-digit	"Phone number must be 10 digits."	400
password	Required	"Password is required."	400



Property Name	Validation	Error Message	Status Code
role	Must be "admin" or "customer"	Role must be 'admin' or 'customer'."	400

Responses	Status Code	Body
Success	201	{ "message": "User registered successfully." }
Validation Error	400	{ "errors": [ { "field": "email", "message": "A valid and unique email is required." } ] }
Conflict	409	{ "message": "User with this email already exists." }

## 2. User Login

### /api/v1/auth/login

#### Method Payload

POST { "email": "john@example.com", "password": "Pass@123" }

Property Name	Validation	Error Message	Status Code
email	Required	"Email is required."	400
password	Required	"Password is required."	400

Responses	Status Code	Body
Success	200	{ "accessToken": "jwt", "role": "customer" }
Invalid Credentials	401	{ "message": "Invalid credentials." }

## 3. Add Vehicle

### /api/v1/vehicles

#### Method Payload

POST multipart/form-data with fields: { "make": "Toyota", "model": "Corolla", "registrationNumber": "ABC1234" } and file:imageFile

Property Name	Validation	Error Message	Status Code
imageFile	Required, jpg/png, max 2MB	"Image must be in .jpg or .png format and less than 2MB."	400
registrationNumber	Required, Unique	"Unique registration number per user is required."	400

Responses	Status Code	Body
Success	201	{ "message": "Vehicle added successfully.", "imageUrl": "/Uploads/car.png", "vehicleId": "<unique_vehicleId>" }
Validation Error	400	{ "errors": [ { "field": "imageFile", "message": "Image must be in .jpg or .png format." } ] }
Forbidden	403	{ "message": "You are not authorized to perform this action." }
Conflict	409	{ "message": "Duplicate registration number." }

## 4. Get Vehicles (Own)

### /api/v1/vehicles

#### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "vehicles": [ { "id": "...", "make": "...", "model": "...", "registrationNumber": "...", "imageUrl": "..." }, ... ] }
Forbidden	403	{ "message": "You are not authorized to perform this action." }

## 5. Get Vehicle by ID

### /api/v1/vehicles/{id}

#### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "vehicle": { "id": "...", "make": "...", "model": "...", "registrationNumber": "...", "imageUrl": "..." } }
Forbidden	403	{ "message": "Unauthorized access." }
Not Found	404	{ "message": "Entity not found." }

## 6. Update Vehicle

/api/v1/vehicles/{id}

### Method Payload

PUT multipart/form-data with fields: { "make": "Honda", "model": "Civic", "registrationNumber": "XYZ123" } and optional file:imageFile

Property Name	Validation	Error Message	Status Code
imageFile	jpg/png, max 2MB	"Image must be in .jpg or .png format and less than 2MB."	400

Responses	Status Code	Body
Success	204	(empty body)
Validation Error	400	{ "errors": [ { "field": "imageFile", "message": "Image must be in .jpg or .png format." } ] }
Forbidden	403	{ "message": "Unauthorized access." }
Not Found	404	{ "message": "Entity not found." }

## 7. Delete Vehicle

/api/v1/vehicles/{id}

### Method Payload

DELETE -

Responses	Status Code	Body
Success	204	(empty body)
Forbidden	403	{ "message": "Unauthorized access." }
Not Found	404	{ "message": "Entity not found." }

## 8. Book Service

/api/v1/bookings

### Method Payload

POST { "vehicleId": "guid", "bookingDate": "2025-05-20", "serviceType": "Premium" }

Property Name	Validation	Error Message	Status Code
bookingDate	Future date, not Sunday	"Booking date must be a future date and not Sunday."	422
serviceType	Must be Basic/Premium/Full	"Invalid service type."	422

Responses	Status Code	Body
Success	201	{ "message": "Service booked successfully with charges applied.", "bookingId": "..." }
Validation Error	422	{ "errors": [ { "field": "bookingDate", "message": "Booking date must be a future date and not Sunday." } ] }
Forbidden	403	{ "message": "You are not authorized to perform this action." }

## 9. Get Own Bookings

/api/v1/bookings/customer

### Method Payload

GET -

Responses	Status Code	Body
Success	200	{ "bookings": [ { "id": "...", "vehicleId": "...", "serviceType": "...", "bookingDate": "...", ... } ] }
Forbidden	403	{ "message": "You are not authorized to perform this action." }

## 10. Get All Bookings

/api/v1/bookings

### Method Payload

GET -

**Responses Status Code Body**

Success	200	{ "bookings": [ { "id": "...", "vehicleId": "...", "serviceType": "...", "bookingDate": "...", ... } ] }
Forbidden	403	{ "message": "You are not authorized to perform this action." }

## 11. Get Booking by ID

/api/v1/bookings/{id}

**Method Payload**

GET -

**Responses Status Code Body**

Success	200	{ "booking": { "id": "...", "vehicleId": "...", "serviceType": "...", "bookingDate": "...", ... } }
Forbidden	403	{ "message": "Unauthorized access." }
Not Found	404	{ "message": "Entity not found." }

## 12. Cancel Booking (Soft Delete)

/api/v1/bookings/{id}

**Method Payload**

DELETE -

**Responses Status Code Body**

Success	204	(empty body)
Forbidden	403	{ "message": "Unauthorized access." }
Not Found	404	{ "message": "Entity not found." }

**Online Reference**

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity-api-authorization?view=aspnetcore-5.0>

**.NET Core Web API****WEB API (old)****Authentication And Authorization (WEBAPI)(old)****FullStackDevelopment\_With\_Dotnet\_AND\_Angular**

# Module 17

## What to learn

### Introduction to Authentication and Authorization

- Overview and key concepts

- Importance in modern application security

### Using JSON Web Tokens (JWT)

- What are JWTs?

- Benefits of using JWTs for authentication

- Implementation in .NET applications

### Real-Time Use Case: Authentication and Authorization

- Practical example of implementing authentication

- Role-based authorization scenarios

- Read the Token from the Authorization Header

- Read Claims from the Token

- Extract Claims from the Validated Token

- Best practices and troubleshooting

## Practice Exercise

No practice exercise

## Assignment Exercise

No assignment exercise

## Online Reference

No online Reference

## .NET Core Web API

### WEB API (old)

### Authentication And Authorization (WEBAPI)(old)

### FullStackDevelopment\_With\_Dotnet\_AND\_Angular

# Module 18

## What to learn

### Introduction to Entity Framework Core

- What is Entity Framework Core?

- Setting up EF Core in a .NET Core project

- Creating DbContext and DbSet

- Hands-on: Add EF Core to the "Books" API and configure a database.

### Code-First Approach

- Code-first migrations

- Seeding the database

- Updating the database schema

- Hands-on: Create and migrate tables for the "Books" API.

### Querying Data

- LINQ basics

- Async queries with EF Core

- Query filtering, sorting, and pagination

- Hands-on: Add filtering, sorting, and pagination to the "Books" API.

### Repository Pattern

- Why use the repository pattern?

- Creating a repository and unit of work

- Hands-on: Refactor the "Books" API to use the repository pattern.

### Error Handling and Logging

- Global exception handling

- Using ILogger for logging

- Implementing custom error responses

- Hands-on: Add centralized error handling to the "Books" API.

## Practice Exercise

### Practice 1

## Understanding Entity Framework Core

Explain what Entity Framework Core is and how it differs from previous versions of Entity Framework. Discuss its advantages in modern .NET Core applications.

### Practice 2

## Setting Up EF Core

Walk through the steps to set up Entity Framework Core in a new .NET Core project. Include details on necessary NuGet packages and configuration settings.

### Practice 3

## Creating DbContext and DbSet

Define a simple `DbContext` class for a 'Books' API. Include at least two `DbSet` properties for entities such as `Book` and `Author`. Explain the purpose of each.

### Practice 4

## Code-First Migrations

Describe the process of creating and applying code-first migrations in EF Core. What commands are used, and what files are generated during this process?

### Practice 5

## LINQ Basics

Provide examples of basic LINQ queries that can be used to retrieve data from a `DbSet` in EF Core. Explain how these queries can be used to filter and sort data.

### Practice 6

## Implementing the Repository Pattern

Discuss the repository pattern and its benefits in managing data access. Provide a brief outline of how to create a repository for the 'Books' API.

### Practice 7

## Error Handling and Logging

Explain the importance of error handling and logging in a Web API. Describe how to implement global exception handling and use `ILogger` for logging errors.

## Assignment Exercise

### Assignment 1

## Books API Development Assignment

Develop a 'Books' API that utilizes Entity Framework Core for data access. The API should support the following functionalities:

**Entity Framework Core Setup:** Set up EF Core in your project and create a `DbContext` for managing books and authors.

**Code-First Approach:** Implement code-first migrations to create the necessary database schema for books and authors. Seed the database

with initial data.

**Querying Data:** Implement endpoints for retrieving books with filtering, sorting, and pagination capabilities. Use LINQ to query the data.

**Repository Pattern:** Refactor your data access code to use the repository pattern, ensuring separation of concerns.

**Error Handling and Logging:** Implement global exception handling and logging using `ILogger`. Ensure that meaningful error messages are returned to the client.

**Functional Flow:** Ensure that all outputs are displayed on the user interface (UI) only, not in the console. The API should be able to handle requests and return appropriate responses based on the operations performed.

**Business Logic:** Clearly outline any specific rules or conditions for data handling and processing, such as validation rules for book entries and author associations.

Online Reference

No online Reference

**.NET Core Web API**

**WEB API (old)**

**Authentication And Authorization (WEBAPI)(old)**

**FullStackDevelopment\_With\_Dotnet\_AND\_Angular**