CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY AND RESEARCH
DEPARTMENT OF COMPUTER ENGINEERING
ACADEMIC YEAR: 2025-26

Subject Name: .Net Core Programming **Semester: 5th (BTech)**

Subject Code: CE387/CSE307 **Academic years: 2025-26**

Practical List

Sr. No.	Practical	Lab Hrs.	CO
	PART-1 .NET Frameworks and C# Concept		
1.	Design a console-based application in C# to track employee attendance within an organization. The system should allow HR personnel to record daily attendance, display all attendance entries, and calculate the number of days an employee was present. A user-friendly console interface should manage all interactions, with data stored in-memory during runtime Requirements and Features: 1. **Design – EmployeeAttendance** o Create a class named EmployeeAttendance to represent the structure of a single attendance record. o Properties: ▪ EmployeeId (int) ▪ EmployeeName (string) ▪ Date (DateTime) ▪ IsPresent (bool) o Use **constructors** and **encapsulation** (with getters/setters) to manage property access. 2. **Data Storage and Business Logic Layer** o Maintain attendance records using a List<EmployeeAttendance>.	2	1

	○ Define a separate class (e.g., AttendanceManager) that contains methods to: ▪ Add a new record ▪ Display all records ▪ Calculate total present days for a specific employee ○ Follow **Single Responsibility Principle** by separating data operations from user interaction. 3. **Console Interface and User Interaction** ○ Develop a **menu-driven interface** using do-while and switch-case. ○ Options to implement: ▪ Record a new attendance entry ▪ Display all attendance records ▪ View total present days by employee ID ▪ Exit the program 4. **Input Handling and Validation** ○ Use try-catch blocks and conditional checks to: ▪ Validate date input format (DateTime.TryParse) ▪ Ensure numeric values for Employee ID ▪ Accept only true/false or Y/N for presence ○ Display appropriate messages for invalid inputs. Sample Console Output: ===== Employee Attendance Tracker ===== 1. Add Attendance Record 2. View All Attendance Records 3. Get Total Present Days by Employee ID4. Exit Enter your choice:		
	PART-2. ASP.NET		
2.	The **Faculty Profile Management System** is a web-based application designed to streamline the process of recording and managing academic and professional details of faculty members in an educational institution. The system provides an intuitive user interface where users can input, review, and maintain essential profile information in a structured format. All modules of the application are integrated within a consistent layout to simulate a real-world enterprise system. The solution ensures form validation, secure access, automated identifier generation, and smooth navigation across different functional pages. Designed to operate entirely without database dependency, this system leverages memory-based data	4	1

handling and logical structures to simulate record storage and access behavior effectively.

I: Faculty Profile Form

Design and implement an interactive form that captures essential details about faculty members for institutional records.

Fields to Capture:

- Full Name
- Employee ID (Auto-generated)
- Department
- Highest Qualification
- Specialization
- Experience (in years)
- Email ID
- Mobile Number

Implementation:

- The form will be developed using an **ASP.NET Web Form (.aspx)** and its associated code-behind file.
- Standard web controls like **textboxes, dropdowns, and radio buttons** will be used to capture inputs.
- **ASP.NET validation controls** will be applied to ensure correctness, including checks for required fields, valid email/mobile formats, and a proper range for numeric values.
- Faculty data entered will be **temporarily stored in session** to preserve user input across multiple pages or refresh actions.
- **ViewState** will be used to retain control-level values during postbacks.
- A **unique Employee ID** will be automatically generated based on the faculty's initials and system timestamp to ensure uniqueness.

II: Master Page Integration

Ensure a consistent, professional, and navigable layout across all system pages using a master page structure.

Implementation:

- A single **Master Page** will be created to define a uniform layout for the application.

	• The layout will include:
	  ◦ A **header** with the institute's logo and title
	  ◦ A **navigation menu** for module switching (e.g., profile entry, list view, logout)
	  ◦ A **footer** with dynamic content such as the current year and contact details
	  ◦ A **welcome banner** displaying the logged-in user's name
	• The master page will **dynamically highlight the active page** in the navigation menu for improved user guidance.
	• **Session validation** will be integrated to restrict access to authenticated users only. Unauthorized users will be redirected to the login page.
	• A **ContentPlaceHolder** will be used to inject page-specific content without altering the master layout, promoting code reuse and consistency.
	
	III: Session-Based Simulated Login
	
	Implement a basic login mechanism that controls access to the system and simulates role-based authorization without requiring a database.
	
	Implementation:
	
	• A **dedicated login page** will allow users to enter a predefined username and password combination to gain access.
	• Authentication will be based on **hardcoded credentials**, simulating a basic admin login experience.
	• Upon successful authentication, the user's identity will be **stored in a session variable** and made available throughout the system.
	• If authentication fails, users will receive an appropriate message prompting them to retry.
	• A **logout option** will be available, which clears the session and redirects the user to the login page, ensuring proper session handling and security.		

PART-3. ADO.NET			
3.	Develop a **Student Management System** using **ASP.NET Web Forms** and **ADO.NET** that allows for efficient student data handling, secure session management, and a consistent user interface. The system must	4	2

	align with modular design principles and maintain data integrity through structured validation and efficient data access methods. System Requirements I. Student Enrollment Management Module Design and implement a **student enrollment form** that captures essential student information and ensures proper validation for mandatory fields including: • Student Name • Roll Number • Course • Enrollment Date • Email ID • Mobile Number Technical requirements: • Implement **ASP.NET validation controls** (e.g., RequiredFieldValidator, RegularExpressionValidator) to enforce input correctness. • Use **Session State** to temporarily store form data to handle navigation without losing entered information. • Use **ADO.NET DataSet** for inserting and updating data. • Use **ADO.NET DataReader** for fast, forward-only retrieval of student records for read-only views. • Auto-generate a unique StudentID using a logic that combines timestamp and student initials to ensure uniqueness across entries. II. Admin Dashboard for Student Data Management Develop an **admin panel** using GridView for complete student data management, including functionalities for viewing, adding, editing, and deleting records. Technical requirements: • Populate GridView using **DataReader** for display operations and **DataSet** for data manipulation. • Enable **paging** in GridView to manage large volumes of student records. • Include editing and deleting capabilities with confirmation dialogs.		

	• Allow data filtering based on specific attributes such as Course or Enrollment Year using dropdowns or text inputs.		
	PART - 4. ASP.NET MVC		
4.	The **Academic Services Department** of a university is in need of a secure and scalable web-based solution to manage **student enrollments across courses**. As a developer in the university's internal development team, you are tasked with building this system using **ASP.NET MVC architecture**. This system will allow academic administrators to register students, assign them to courses, manage enrollment updates, and maintain a clean view of course-wise student listings. 🔷 Functional Requirements: 🔷 Admin Tasks: 1. Add new students with validated details. 2. Create and manage course offerings (course name, code, credits). 3. Assign one or more courses to a student during enrollment. 4. View all student records along with enrolled courses. 5. Edit or delete a student's record. 6. Filter students by course, name, or enrollment year. 7. Navigate through the application using **well-defined routes**. MVC Implementation Strategy: Model Layer: • Define clean domain models: Student, Course, Enrollment • Use **Data Annotations** for constraints (e.g., Required, Range, Email format) • Maintain in-memory collections to simulate a repository Controller Layer: • Create StudentController and CourseController • Implement strongly typed action methods for: o Index, Create, Edit, Delete, Details	4	3

	• Include **custom actions** like:
	  ○ AssignCourses(int studentId)
	  ○ GetStudentsByCourse(string courseCode)
	
	View Layer (Razor):
	
	• Use strongly typed Razor views for all operations
	• Build interactive forms with Html.BeginForm, @Html.EditorFor, etc.
	• Display error messages using @Html.ValidationMessageFor
	• Use foreach to list student-course enrollments in tabular format
	• Apply conditional UI logic (e.g., "Show Courses if Assigned")
	
	Routing & Navigation:
	
	• Define **custom routes** such as:
	  ○ /students/enrolled
	  ○ /student/edit/102
	  ○ /course/assign/CS101
	• Utilize RouteConfig.cs for defining logical URL structures
	• Use **attribute routing** for specific controller methods		

	PART - 5. ASP.NET Core MVC		
5.	**Student Record Manager—**Display Student Data using MVC Architecture **Scenario** You are working as a software developer in the university's IT department. The faculty has requested a web application that allows them to view student records such as ID, name, department, and CGPA. As part of the initial prototype, your task is to create a simple ASP.NET Core MVC application that will display a list of students using hardcoded data (i.e., not connected to a database). This prototype should follow the MVC design pattern: □ **Model**: Define a Student class that encapsulates student attributes including ID, Name, Department, CGPA, and Email. □ **Controller**: Create a controller (StudentController) that initializes and supplies a hardcoded list of Student objects to the view.	2	4

	View: Implement a Razor view to display student records in a structured HTML table with columns for ID, Name, Department, CGPA, and Email. **Tasks** Task 1: Create a New ASP.NET Core MVC Project **Task 2**: Create a Student model class with properties: StudentId, Name, Department, CGPA. **Task 3**: Add a StudentController and define an Index() action method that prepares and passes a list of hardcoded students to the view. **Task 4**: Design a Student/Index.cshtml view to display the list using an HTML table format, properly binding data from the model. **Task 5**: Configure the Startup.cs or Program.cs and appsettings.json (if required) to ensure the default route maps to StudentController -> Index. **Task 6**: Build and run the application to test that the student records are displayed correctly. **Task 7**: • Add a new property Email to the Student model. • Update the controller to include email addresses in the hardcoded list. • Modify the view to include a new column for displaying the email address.		
6.	**ASP.NET Core MVC - CRUD** **Employee Information – CRUD Operations Without Database** **Scenario** You are working as a software developer in the Human Resources (HR) department of a corporate company. The HR team has requested a simple internal web application to manage employee records during the planning phase. Since this is a **prototype**, the data will be stored in-memory, not in	4	4

a database.

Your task is to build a **basic ASP.NET Core MVC application** to perform **CRUD (Create, Read, Update, Delete)** operations on employee information using a simple **List in memory**.

The application must follow the **MVC design pattern**:

- The **Model** will define the structure of the Employee.
- The **Controller** will manage the in-memory list and logic for CRUD operations.
- The **Views** will render user interfaces to interact with the employee data.

Tasks

Task 1: Create a New ASP.NET Core MVC Project

Task 2: Define the Employee Model

Create an Employee class inside the **Models** folder with the following properties:

- Id (int)
- Name (string)
- Department (string)
- Designation (string)
- Email (string)
- Salary (decimal)

Task 3: Create the EmployeeController

Add a new EmployeeController that manages an **in-memory** List<Employee>, declared as a static list or via dependency injection. Implement logic to manipulate this list in memory (without database).

Task 4: Implement CRUD Operations in the Controller

Implement all necessary action methods in the controller:

- Index() – Display the list of employees.
- Details(int id) – Show detailed information of a specific employee.
- Create() – Show form and handle form submission to add a new employee.

	• Edit(int id) – Show form and handle update logic. • Delete(int id) – Confirm and remove an employee from the list. Task 5: Create Razor Views for All Actions Design corresponding Razor views using HTML helpers (@Html.EditorFor, @Html.ValidationMessageFor, etc.) for: • Listing all employees • Creating a new employee • Editing existing employee details • Viewing employee details • Deleting an employee Task 6: Implement Form Validation Use **Data Annotations** in the Employee model to apply validation rules such as: • [Required] for mandatory fields • [EmailAddress] for email validation • [Range] for salary limits Display validation messages in views using built-in Razor syntax. Task 7: Configure Default Route Update the routing configuration in Program.cs to ensure the application launches with EmployeeController and Index action. Task 8: Build and Run the Application Compile and run the application. Validate the correctness of all CRUD operations in the browser using the in-memory list. Task 9: Documentation Capture and submit **screenshots of each operation**: • Create • Read (Index and Details) • Update • Delete		

7.	ASP.NET Core Web API Product Information – CRUD Operations Without Database Scenario You are working as a software developer in a retail company's IT department. The inventory team has requested a lightweight RESTful API to manage product data for testing and internal usage. This is an initial prototype and does not require a database at this stage. Instead, all product data will be stored and manipulated in an in-memory list during runtime. Your task is to build a basic ASP.NET Core Web API that allows external applications (e.g., frontend apps, Postman, mobile clients) to create, read, update, and delete product records through HTTP requests. This project will not include views or UI. It will follow REST principles and use the standard HTTP verbs: ● GET to read data, ● POST to create data, ● PUT to update data, ● DELETE to remove data. Tasks Task 1: Create a New ASP.NET Core Web API Project Task 2: Create the Product Model Define a class Product inside the **Models** folder with properties: • Id (int) • Name (string) • Category (string) • Price (decimal) • Stock (int)	4	4

Task 3: Create the ProductController

- Add a controller named ProductController inside the **Controllers** folder.
- Decorate it with:

```
[ApiController]
[Route("api/[controller]")]
```

Task 4: Define In-Memory Data Store

Declare a static or singleton List<Product> in the controller to simulate an in-memory database.

Task 5: Implement CRUD Endpoints Using HTTP Verbs

- GET /api/product – Return the list of products.
- GET /api/product/{id} – Return a specific product by ID.
- POST /api/product – Add a new product to the list.
- PUT /api/product/{id} – Update an existing product by ID.
- DELETE /api/product/{id} – Remove a product from the list by ID.

Each action should return appropriate **HTTP status codes** such as 200 OK, 201 Created, 400 Bad Request, or 404 Not Found.

Task 6: Add Validation with Data Annotations

Apply data validation rules using **Data Annotations** in the model:

- [Required] for Name and Category
- [Range(0.01, 100000)] for Price
- [Range(0, int.MaxValue)] for Stock

Model validation should be enforced automatically via [ApiController].

Task 7: Test Endpoints Using Swagger or Postman

- Use **Swagger UI** (auto-generated in ASP.NET Core Web API) or **Postman** to:
  - Create (POST) new product entries.
  - Read (GET) the full list or a single product.
  - Update (PUT) a product by ID.
  - Delete (DELETE) a product by ID.

	Take screenshots of each tested endpoint response.		
8.	ASP.NET Core MVC with Entity Framework Core Product Information Manager – CRUD Operations Using MVC and Database Scenario You are working as a software developer in a retail company's IT department. The inventory management team has requested a web-based system to manage product information. The system should allow users to add, view, edit, and delete product records from a database. Your task is to build a basic ASP.NET Core MVC application using Entity Framework Core for database operations. This application will provide a user-friendly interface to manage product information such as Product ID, Name, Category, Price, and Quantity. The application should follow the MVC architecture: ● The Model will define the product structure. ● The Controller will handle CRUD logic and communicate with the database. ● The Views will provide interfaces for user interaction. Tasks: Task 1: Create a New ASP.NET Core MVC Project Task 2: Configure SQL Server and Entity Framework Core • Install the necessary NuGet packages: o Microsoft.EntityFrameworkCore.SqlServer o Microsoft.EntityFrameworkCore.Tools • Configure the **connection string** in appsettings.json. Task 3: Create the Product Model Add a Product class inside the **Models** folder with fields:	4	5

- ProductId (int)
- Name (string)
- Category (string)
- Price (decimal)
- Quantity (int)

Apply **Data Annotations** such as:

- [Required], [StringLength], [Range]

Task 4: Setup the Database Context Class

Create a ProductDbContext class inheriting from DbContext and add DbSet<Product> Products.

Task 5: Apply EF Core Migrations

- Use CLI or Package Manager Console:

```
Add-Migration InitialCreate
Update-Database
```

- Ensure the SQL Server database is created with the Products table.

Task 6: Scaffold Controller and Views

- Use **Scaffolding** to generate:
  - ProductController
  - Razor Views for Index, Create, Edit, Details, and Delete
- Connect them to EF Core through the context.

Task 7: Configure Default Route

Set the default controller and action in Program.cs:

```
app.MapControllerRoute(
    name: "default",
    pattern:
"{controller=Product}/{action=Index}/{id?}");
```

Task 8: Build and Run the Application

- Launch the application in a browser.

	• Verify all CRUD operations (Create, Read, Update, Delete) are working.
	• Data should persist in the SQL Server database.
	
	Task 9: Add Validation to Product Model
	
	• Ensure server-side validation rules are enforced.
	• Display error messages using Razor helpers like:
	
	`@Html.ValidationMessageFor(model => model.Price)`
	
	Task 10: Documentation and Output
	
	• Submit:
	  ○ Screenshots of each CRUD operation.
	  ○ A short summary explaining:
	  ▪ How MVC structure is followed (Model, View, Controller)
	  ▪ How EF Core handles data interaction and migrations.		

9.	**LINQ Operations – Querying and Manipulating In-Memory Collections** **Scenario** You are working as a software developer for a data analytics company. Your manager has asked you to create a prototype that can perform various operations on in-memory collections (like lists of products or employees) to filter, sort, group, and summarize the data. Instead of writing traditional loops and conditionals, you are required to use LINQ (Language Integrated Query) to query and manipulate data using clean, concise, and readable syntax. This practical will help you understand how LINQ works with in-memory collections like List<T> using both query syntax and method syntax. **Tasks:** ● Task 1: Create a New Console Application in C# Named	2	5

	LinqOperationsDemo ● Task 2: Define a Class Product with the appropriate Properties: (Product ID, Name, Category, Price, Stock Quantity) ● Task 3: Create a List of Products and Populate it with at Least 6 Sample Records ● Task 4: Perform the Following LINQ Operations Using Method Syntax: ○ Display all products in the list ○ Filter products where Price > 500 ○ Get products that belong to a specific Category (e.g., "Electronics") ○ Sort products by Name in ascending order ○ Calculate the total stock quantity of all products ○ Find the most expensive product ● Task 5: Perform the Following LINQ Operations Using Query Syntax: ○ Select only product names from the list ○ Group products by Category ○ Count the number of products in each category ○ Find average price of all products ● Task 6: Display the Output of Each Query in the Console with Proper Labels		