Full Stack .NET Training Manual

Section 1 – Table of Contents

Section 1 – Core C# & OOP Concepts

Chapter 1: Introduction to .NET & C#

1.1 What is .NET?
1.2 Evolution of .NET
1.3 What is C#?
1.4 Advantages of C# for Developers
1.5 First Program in C# – "Hello, World!"
1.6 Anatomy of a C# Program
1.7 Comments in C#
1.8 Real-World Case Study – Why Use C#?
1.9 Exercises

Chapter 2: Data Types, Variables, and Operators

2.1 Variables in C#
2.2 Naming Rules for Variables
2.3 Data Types in C#

- Value Types
- Reference Types

2.4 Common Data Types
2.5 Constants & Readonly Variables
2.6 Operators in C#

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

2.7 Real-World Example – Employee Salary Calculator
2.8 Exercises

Chapter 3: Conditional Statements

3.1 Introduction
3.2 The `if` Statement

3.3 The `if-else` Statement
3.4 Nested `if` Statements
3.5 The `switch` Statement
3.6 Ternary Operator (`?:`)
3.7 Real-World Example – ATM Withdrawal Validation
3.8 Exercises

Chapter 4: Loops in C#
4.1 Introduction
4.2 `for` Loop
4.3 `while` Loop
4.4 `do-while` Loop
4.5 `foreach` Loop
4.6 `break` and `continue`
4.7 Real-World Example – Password Retry System
4.8 Exercises

Chapter 5: Arrays & Collections
5.1 Arrays – Basics
5.2 Iterating Over Arrays
5.3 Multi-Dimensional Arrays
5.4 Jagged Arrays (Array of Arrays)
5.5 Collections in C#

- List<T>
- Dictionary<TKey, TValue>
- Queue<T>
- Stack<T>

5.6 Real-World Example – Student Marks with Dictionary
5.7 Exercises

Chapter 6: Methods in C#
6.1 What is a Method?
6.2 Method Syntax
6.3 Calling Methods
6.4 Types of Methods
6.5 Pass by Value vs Pass by Reference
6.6 Optional & Named Parameters
6.7 Real-World Example – Bank Account Methods
6.8 Exercises

Chapter 7: Object-Oriented Programming (OOP) Concepts

7.1 Introduction

7.2 Four Pillars of OOP

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

7.3 Classes & Objects

7.4 Constructors

7.5 Encapsulation – Properties

7.6 Inheritance

7.7 Polymorphism

- Compile-Time (Method Overloading)
- Run-Time (Method Overriding)

7.8 Abstraction with Interfaces & Abstract Classes

7.9 Real-World Example – E-commerce System

7.10 Exercises

Chapter 8: Exception Handling & Debugging

8.1 What is Exception Handling?

8.2 Syntax of try-catch-finally

8.3 Common Exceptions

8.4 Throwing Exceptions

8.5 Custom Exceptions

8.6 Debugging Techniques in Visual Studio

8.7 Real-World Example – Banking Transaction

8.8 Exercises

Chapter 9: Exercises, Mini Projects & Case Studies

9.1 Mini Project – Library Management System (Console App)

9.2 Practice Problems

9.3 Case Studies – ATM System, Student Management, E-commerce

Section 2 – Table of Contents (Index)

Section 2 – ASP.NET Core & Web Development

Chapter 1: Introduction to ASP.NET Core & MVC Architecture

1.1 What is ASP.NET Core?

1.2 Features of ASP.NET Core

1.3 Differences between ASP.NET Framework & ASP.NET Core

1.4 Understanding the MVC Architecture (Model-View-Controller)
1.5 Benefits of Using MVC
1.6 Real-World Example – Simple Web App
1.7 Exercises

Chapter 2: Project Structure in ASP.NET Core, Middleware & Pipeline
2.1 ASP.NET Core Project Structure Overview
2.2 Program.cs and Startup.cs Explained
2.3 Dependency Injection in Startup
2.4 Middleware – Concept & Usage
2.5 Request Processing Pipeline
2.6 Writing Custom Middleware
2.7 Exercises

Chapter 3: Controllers & Action Methods
3.1 Introduction to Controllers
3.2 Creating a Controller
3.3 Action Methods Explained
3.4 Returning Different Result Types (View, JSON, Redirect)
3.5 Routing in ASP.NET Core
3.6 Attribute Routing
3.7 Exercises

Chapter 4: Views, Razor Syntax & Layout Pages
4.1 Introduction to Views
4.2 Razor Syntax Basics (@ symbol, code blocks, expressions)
4.3 Layout Pages & Partial Views
4.4 Using Sections and RenderBody
4.5 ViewData, ViewBag, and TempData
4.6 Strongly-Typed Views
4.7 Exercises

Chapter 5: Models & ViewModels, Data Binding
5.1 Introduction to Models
5.2 Creating a Model Class
5.3 Understanding ViewModels
5.4 Passing Data from Controller to View
5.5 Model Binding Basics
5.6 Validation using Data Annotations
5.7 Exercises

Chapter 6: Form Handling & Validation
6.1 Creating HTML Forms in Razor Views

6.2 Binding Form Data to Models
6.3 Server-Side Validation
6.4 Client-Side Validation
6.5 Validation Summary & Validation Messages
6.6 Real-World Example – Registration Form
6.7 Exercises

Chapter 7: Dependency Injection (DI) in ASP.NET Core
7.1 What is Dependency Injection?
7.2 Built-in DI in ASP.NET Core
7.3 Registering Services (Transient, Scoped, Singleton)
7.4 Injecting Services into Controllers
7.5 Real-World Example – Logging Service
7.6 Exercises

Chapter 8: Introduction to Entity Framework Core (EF Core)
8.1 What is EF Core?
8.2 Benefits of Using EF Core
8.3 DbContext & DbSet Explained
8.4 Configuring Connection Strings
8.5 Code-First Approach Overview
8.6 Database-First Approach Overview
8.7 Exercises

Chapter 9: Database Operations with EF Core
9.1 CRUD Operations – Create, Read, Update, Delete
9.2 LINQ Queries
9.3 Using Migrations
9.4 Relationships (One-to-One, One-to-Many, Many-to-Many)
9.5 Real-World Example – Employee Management App
9.6 Exercises

Chapter 10: Final Mini Project
10.1 Project Overview
10.2 Designing Models, Views, and Controllers
10.3 Implementing CRUD Operations
10.4 Validation & Exception Handling
10.5 Deployment Overview
10.6 Exercises & Enhancements

Section 3 – Table of Contents (Index)

Section 3 – Angular Basics

Chapter 1: Introduction to Angular
1.1 What is Angular?
1.2 History and Evolution of Angular
1.3 Angular vs AngularJS
1.4 Features of Angular
1.5 Benefits of Using Angular for Frontend Development
1.6 Real-World Examples of Angular Applications
1.7 Exercises

Chapter 2: Setting Up Angular Development Environment
2.1 Installing Node.js and npm
2.2 Installing Angular CLI
2.3 Creating a New Angular Project
2.4 Project Structure Overview
2.5 Running Angular Application
2.6 Exercises

Chapter 3: Angular Components
3.1 Introduction to Components
3.2 Creating Components Using CLI
3.3 Component Decorator and Metadata
3.4 Template, Style, and Selector
3.5 Lifecycle Hooks of Components
3.6 Nested Components
3.7 Exercises

Chapter 4: Data Binding in Angular
4.1 Introduction to Data Binding
4.2 Types of Data Binding

- Interpolation
- Property Binding
- Event Binding
- Two-Way Binding

4.3 Real-World Examples of Data Binding
4.4 Exercises

Chapter 5: Angular Directives

5.1 Introduction to Directives
5.2 Structural Directives (`*ngIf`, `*ngFor`)
5.3 Attribute Directives (`[ngStyle]`, `[ngClass]`)
5.4 Custom Directives
5.5 Exercises

Chapter 6: Angular Services and Dependency Injection

6.1 What is a Service in Angular?
6.2 Creating and Using Services
6.3 Dependency Injection in Angular
6.4 Singleton Services
6.5 Exercises

Chapter 7: Angular Routing and Navigation

7.1 Introduction to Routing
7.2 Configuring Routes in Angular
7.3 RouterLink and RouterOutlet
7.4 Route Parameters
7.5 Nested Routes
7.6 Exercises

Chapter 8: Angular Forms

8.1 Introduction to Forms in Angular
8.2 Template-Driven Forms
8.3 Reactive Forms
8.4 Form Validation (Built-in Validators & Custom Validators)
8.5 Exercises

Chapter 9: Angular HTTP Client and APIs

9.1 Introduction to HTTP Client
9.2 Making GET, POST, PUT, DELETE Requests
9.3 Observables and RxJS Basics
9.4 Handling HTTP Responses and Errors
9.5 Real-World Example – Connecting to a REST API
9.6 Exercises

Chapter 10: Final Mini Project
10.1 Project Overview – Angular CRUD App
10.2 Creating Components and Services
10.3 Implementing Routing
10.4 Form Handling and Validation
10.5 Integrating with Backend API (ASP.NET Core)
10.6 Exercises & Enhancements

Section 4 – Table of Contents (Index)

Section 4 – Advanced Topics / Full Stack Integration

Chapter 1: Introduction to Full Stack Integration
1.1 What is Full Stack Development
1.2 Overview of Frontend (Angular) and Backend (ASP.NET Core) Integration
1.3 Benefits of Full Stack Applications
1.4 Architecture of Full Stack Apps (Client-Server-Database)
1.5 Exercises

Chapter 2: Setting Up Full Stack Environment
2.1 ASP.NET Core API Project Setup
2.2 Angular Frontend Project Setup
2.3 Connecting Angular Frontend with ASP.NET Core API
2.4 CORS Configuration
2.5 Exercises

Chapter 3: Advanced Angular Concepts for Full Stack
3.1 Lazy Loading Modules
3.2 Route Guards (AuthGuard, CanActivate)
3.3 Angular Interceptors for HTTP Requests
3.4 State Management Basics (BehaviorSubject, Services)
3.5 Exercises

Chapter 4: Advanced ASP.NET Core Concepts
4.1 Repository Pattern
4.2 Unit of Work Pattern
4.3 Dependency Injection in Advanced Scenarios

4.4 Middleware & Exception Handling
4.5 Logging and Monitoring
4.6 Exercises

Chapter 5: Authentication & Authorization
5.1 Introduction to Authentication and Authorization
5.2 JWT (JSON Web Token) Authentication
5.3 Role-Based Authorization
5.4 Token Validation in Angular and ASP.NET Core
5.5 Exercises

Chapter 6: CRUD Operations – Full Stack Example
6.1 Creating Models in ASP.NET Core
6.2 Implementing API Controllers with CRUD Endpoints
6.3 Connecting Angular Services to API
6.4 Displaying Data in Angular Components
6.5 Exercises

Chapter 7: File Upload and Download in Full Stack Applications
7.1 Uploading Files from Angular to ASP.NET Core
7.2 Downloading Files from Backend
7.3 Handling File Validation and Size Limits
7.4 Exercises

Chapter 8: Advanced Data Handling
8.1 Pagination in Angular and ASP.NET Core
8.2 Sorting and Filtering Data
8.3 Search Functionality Implementation
8.4 Exercises

Chapter 9: Deployment of Full Stack Applications
9.1 Preparing ASP.NET Core Application for Production
9.2 Building Angular Project for Production
9.3 Hosting Full Stack Application on IIS / Azure / Docker
9.4 Exercises

Chapter 10: Final Full Stack Mini Project
10.1 Project Overview – Online Product Management System
10.2 Setting up Backend (ASP.NET Core API + Database)
10.3 Creating Angular Frontend with Routing and Forms
10.4 Implementing Authentication and Authorization
10.5 Integrating CRUD, Search, and Pagination
10.6 Deployment Steps
10.7 Exercises and Enhancements

Section 1 – Core C# & OOP Concepts

Chapter 1: Introduction to .NET & C#

1.1 What is .NET?

.NET is a free, open-source, cross-platform framework developed by Microsoft. It provides:

- A **runtime environment** for executing applications (CLR – Common Language Runtime).
- A **Base Class Library (BCL)** that includes thousands of reusable classes for I/O, networking, collections, threading, security, and more.
- Support for building multiple types of applications:
    - Desktop (Windows Forms, WPF)
    - Web (ASP.NET Core, Blazor)
    - Mobile (Xamarin, MAUI)
    - Cloud (Azure)
    - Gaming (Unity uses C#)
    - IoT & AI

1.2 Evolution of .NET

- **2002:** .NET Framework 1.0 introduced (Windows-only).
- **2016:** .NET Core introduced (cross-platform).
- **2020:** Microsoft unified the platform under **.NET 5 and beyond** (replacing ".NET Core" and ".NET Framework").
- **Today:** Latest versions like **.NET 8** support modern cloud-native and AI-powered applications.

Diagram – .NET Evolution:

```
.NET Framework (2002) → .NET Core (2016) → .NET 5/6/7/8 (2020+)
```

1.3 What is C#?

- C# (pronounced “C-Sharp”) is a modern, **object-oriented, type-safe programming language**.
- Designed by **Anders Hejlsberg** at Microsoft (same person who created Turbo Pascal & Delphi).
- Runs on **CLR (Common Language Runtime)** which manages memory, security, and exception handling.
- C# syntax is influenced by C, C++, and Java but simplifies many features.

Key Features of C#:

- Object-Oriented (supports Classes, Inheritance, Polymorphism).
- Strongly Typed & Type-Safe (avoids unsafe memory operations).
- Automatic Garbage Collection.
- Rich Standard Library.
- Supports Asynchronous Programming (`async` and `await`).
- Language Integrated Query (LINQ).
- Cross-Platform (via .NET Core).

1.4 Advantages of C# for Developers

- **Productivity:** Easy-to-read syntax with fewer errors.
- **Versatility:** Can build web, desktop, mobile, cloud, and AI solutions.
- **Community Support:** Millions of developers and rich learning resources.
- **Integration with Microsoft Ecosystem:** Azure, Office 365, SQL Server.

1.5 First Program in C# – "Hello, World!"

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Explanation:

- `using System;` → Imports the **System namespace**, which contains basic classes like Console.
- `class Program` → Defines a class. In C#, all code must be inside a class.
- `static void Main()` → Entry point of every C# application.
- `Console.WriteLine("Hello, World!");` → Prints text to the console.

Output:

```
Hello, World!
```

1.6 Anatomy of a C# Program

```
using System;

namespace MyFirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to C# Training!");
        }
    }
}
```

Parts Explained:

- **Namespace:** Groups related classes.
- **Class:** Blueprint for objects.
- **Main Method:** Starting point of execution.
- **Statements:** Instructions to perform tasks.

1.7 Comments in C#

- **Single-line comment:** `// This is a comment`
- **Multi-line comment:**

```
/* This is a
   multi-line comment */
```

- **XML Documentation Comment:**

```
/// <summary>
/// This method prints a greeting
/// </summary>
```

1.8 Real-World Case Study – Why Use C#?

Scenario: A retail company wants to build an ERP system.

- **Frontend:** Angular
- **Backend:** ASP.NET Core (C#)
- **Database:** SQL Server
- **Cloud:** Azure

C# is chosen because:

- Works seamlessly with ASP.NET Core.
- Has built-in security features.
- Scales well for enterprise solutions.

1.9 Exercises

1. Write a C# program to display your name and age.
2. Modify the "Hello World" program to print three different lines.
3. Add single-line, multi-line, and XML comments in your code.

Chapter 2: Data Types, Variables, and Operators

2.1 Variables in C#

A **variable** is a name for a memory location that stores data.

Syntax:

```
datatype variableName = value;
```

Example:

```
int age = 25;
string name = "Rahul";
```

2.2 Naming Rules

- Must start with a letter or underscore.
- Cannot contain spaces or special characters.
- Case-sensitive.
- Cannot use reserved keywords (`class`, `int`, etc.).

2.3 Data Types in C#

Value Types

Stored in the **stack memory**, directly hold data.
Examples: `int`, `double`, `char`, `bool`, `struct`, `enum`.

Reference Types

Stored in the **heap memory**, store references to data.
Examples: `class`, `string`, `arrays`, `object`.

2.4 Common Data Types

Data Type	Size	Example	Description
int	4 bytes	100	Integer numbers
double	8 bytes	99.99	Decimal numbers
char	2 bytes	'A'	Single character
bool	1 bit	true/false	Logical values
string	Varies	"Hello"	Sequence of characters

2.5 Constants & Readonly Variables

Constant: Value fixed at compile-time.

```
const double Pi = 3.14159;
```

Readonly: Value can be set only once, at runtime.

```
readonly string companyName = "Tech Solutions";
```

2.6 Operators in C#

Arithmetic Operators

```
int a = 10, b = 5;
Console.WriteLine(a + b);  // 15
Console.WriteLine(a - b);  // 5
```

Relational Operators

```
Console.WriteLine(a > b);  // True
Console.WriteLine(a == b); // False
```

Logical Operators

```
bool x = true, y = false;
Console.WriteLine(x && y); // False
```

Assignment Operators

```
int x = 10;
x += 5; // 15
```

2.7 Real-World Example – Employee Salary Calculator

```
class Employee
{
    public string Name;
    public double Salary;
    public double Bonus;
```

```
    public double GetTotalSalary()
    {
        return Salary + Bonus;
    }
}
```

2.8 Exercises

1. Create variables for student details (Name, Age, Marks). Print them.
2. Write a program to calculate simple interest.
3. Declare a constant for Pi and calculate area of a circle.

Chapter 3: Conditional Statements

3.1 Introduction

Conditional statements allow a program to **make decisions** based on conditions (true/false).

3.2 The `if` Statement

Syntax:

```
if (condition)
{
    // code to execute if condition is true
}
```

Example:

```
int age = 20;
if (age >= 18)
{
    Console.WriteLine("You are eligible to vote.");
}
```

3.3 The `if-else` Statement

```
if (age >= 18)
{
    Console.WriteLine("You can vote.");
}
else
{
    Console.WriteLine("You are too young to vote.");
}
```

3.4 Nested `if`

```
int marks = 85;
if (marks >= 90)
    Console.WriteLine("Grade A+");
else if (marks >= 75)
    Console.WriteLine("Grade A");
else if (marks >= 50)
    Console.WriteLine("Grade B");
else
    Console.WriteLine("Fail");
```

3.5 `switch` Statement

```
int day = 3;

switch (day)
{
    case 1: Console.WriteLine("Monday"); break;
    case 2: Console.WriteLine("Tuesday"); break;
    case 3: Console.WriteLine("Wednesday"); break;
    default: Console.WriteLine("Invalid day"); break;
}
```

3.6 Ternary Operator (? :)

```
int age = 20;
string result = (age >= 18) ? "Adult" : "Minor";
Console.WriteLine(result);
```

3.7 Real-World Example – ATM Withdrawal Validation

```
int balance = 5000;
int withdraw = 2000;

if (withdraw <= balance)
{
    Console.WriteLine("Transaction Successful!");
    balance -= withdraw;
}
else
{
    Console.WriteLine("Insufficient Balance!");
}
```

3.8 Exercises

1. Write a program to check if a number is **even or odd**.
2. Write a program to determine if a year is a **leap year**.
3. Use `switch` to print the name of a month.

Chapter 4: Loops in C#

4.1 Introduction

Loops allow execution of a block of code **multiple times** until a condition is met.

4.2 for Loop

```
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine("Iteration: " + i);
}
```

4.3 while Loop

```
int i = 1;
while (i <= 5)
{
    Console.WriteLine("Iteration: " + i);
    i++;
}
```

4.4 do-while Loop

```
int i = 1;
do
{
    Console.WriteLine("Iteration: " + i);
    i++;
} while (i <= 5);
```

4.5 foreach Loop (Collections)

```
string[] fruits = { "Apple", "Banana", "Cherry" };

foreach (string fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

4.6 break and continue

```
for (int i = 1; i <= 10; i++)
{
    if (i == 5) break; // exit loop
    Console.WriteLine(i);
}

for (int i = 1; i <= 10; i++)
{
    if (i == 5) continue; // skip iteration
    Console.WriteLine(i);
}
```

4.7 Real-World Example – Password Retry System

```
string correctPassword = "admin123";
string input;
int attempts = 0;

do
{
    Console.Write("Enter password: ");
    input = Console.ReadLine();
    attempts++;

    if (input == correctPassword)
    {
        Console.WriteLine("Login successful!");
        break;
    }
    else
    {
        Console.WriteLine("Incorrect password. Try again.");
    }

} while (attempts < 3);

if (attempts == 3 && input != correctPassword)
{
    Console.WriteLine("Account locked.");
```

```
}
```

4.8 Exercises

1. Print multiplication table of any number using a loop.
2. Print first 20 Fibonacci numbers.
3. Write a program to find factorial of a number using a loop.

Chapter 5: Arrays & Collections

5.1 Arrays – Basics

An **array** is a collection of fixed-size, same-type elements.

Syntax:

```
datatype[] arrayName = new datatype[size];
```

Example:

```
int[] numbers = new int[5] {1, 2, 3, 4, 5};
```

5.2 Iterating Over Arrays

```
int[] marks = { 85, 90, 78, 88, 76 };

foreach (int mark in marks)
{
    Console.WriteLine(mark);
}
```

5.3 Multi-Dimensional Arrays

```
int[,] matrix = { {1, 2}, {3, 4}, {5, 6} };
Console.WriteLine(matrix[2,1]); // 6
```

5.4 Jagged Arrays (Array of Arrays)

```
int[][] jagged = new int[2][];
jagged[0] = new int[] {1, 2, 3};
jagged[1] = new int[] {4, 5};

foreach (int[] arr in jagged)
{
    foreach (int num in arr)
        Console.Write(num + " ");
}
```

5.5 Collections in C#

Collections are **dynamic containers** (size can grow/shrink).

List<T>

```
List<string> cities = new List<string>();
cities.Add("Delhi");
cities.Add("Mumbai");
cities.Add("Chennai");

foreach (string city in cities)
    Console.WriteLine(city);
```

Dictionary<TKey, TValue>

```
Dictionary<int, string> students = new Dictionary<int, string>();
students.Add(1, "Rahul");
students.Add(2, "Neha");

foreach (var kvp in students)
```

```
    Console.WriteLine("Roll: " + kvp.Key + " Name: " + kvp.Value);
```

Queue<T>

FIFO structure.

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("Task1");
queue.Enqueue("Task2");
Console.WriteLine(queue.Dequeue()); // Task1
```

Stack<T>

LIFO structure.

```
Stack<string> stack = new Stack<string>();
stack.Push("A");
stack.Push("B");
Console.WriteLine(stack.Pop()); // B
```

5.6 Real-World Example – Student Marks with Dictionary

```
Dictionary<string, int> marks = new Dictionary<string, int>()
{
    {"Rahul", 85},
    {"Neha", 92},
    {"Arjun", 78}
};

foreach (var item in marks)
{
    Console.WriteLine(item.Key + " : " + item.Value);
}
```

5.7 Exercises

1. Create an array of 10 numbers and print them in reverse order.
2. Store 5 employee names using a `List<string>`.

3. Use a `Dictionary` to map roll numbers to student names.

Chapter 6: Methods in C#

6.1 What is a Method?

A **method** is a block of code that performs a specific task.

- Increases **code reusability**.
- Improves **readability and maintainability**.

6.2 Method Syntax

```
returnType MethodName(parameters)
{
    // method body
}
```

Example:

```
class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

6.3 Calling Methods

```
Calculator calc = new Calculator();
int sum = calc.Add(5, 10);
Console.WriteLine(sum); // 15
```

6.4 Types of Methods

- **Instance Methods** → Belong to an object.
- **Static Methods** → Belong to the class, not objects.
- **Void Methods** → No return value.
- **Parameterized Methods** → Accept parameters.
- **Method Overloading** → Same method name with different parameter types.

Example – Static Method:

```
class Utility
{
    public static void Greet(string name)
    {
        Console.WriteLine("Hello, " + name);
    }
}

Utility.Greet("Neha");
```

6.5 Pass by Value vs Pass by Reference

```
void Increment(int x)
{
    x++;
}

int num = 5;
Increment(num);
Console.WriteLine(num); // Still 5

void Increment(ref int x)
{
    x++;
}

int num = 5;
Increment(ref num);
Console.WriteLine(num); // Now 6
```

6.6 Optional & Named Parameters

```
void PrintDetails(string name, int age = 18)
{
    Console.WriteLine(name + " " + age);
}

PrintDetails("Rahul");           // Rahul 18
PrintDetails("Neha", 22);        // Neha 22
```

6.7 Real-World Example – Bank Account Methods

```
class BankAccount
{
    public double Balance = 0;

    public void Deposit(double amount)
    {
        Balance += amount;
    }

    public void Withdraw(double amount)
    {
        if (amount <= Balance)
            Balance -= amount;
        else
            Console.WriteLine("Insufficient funds");
    }

    public double GetBalance()
    {
        return Balance;
    }
}
```

6.8 Exercises

1. Write a method to calculate factorial of a number.
2. Create a method that returns the maximum of three numbers.

3. Write a program to demonstrate method overloading.

Chapter 7: Object-Oriented Programming (OOP) Concepts

7.1 Introduction

OOP is a programming paradigm based on the concept of **objects** that contain **data (fields)** and **behavior (methods)**.

7.2 Four Pillars of OOP

1. **Encapsulation** → Wrapping data & methods inside a class.
2. **Inheritance** → Reuse existing classes in new ones.
3. **Polymorphism** → Same method name, different behavior.
4. **Abstraction** → Hiding implementation details, showing only functionality.

7.3 Classes & Objects

Class: Blueprint of an object.
Object: Instance of a class.

```
class Car
{
    public string Brand;
    public void Drive()
    {
        Console.WriteLine(Brand + " is driving...");
    }
}

Car myCar = new Car();
myCar.Brand = "BMW";
myCar.Drive();
```

7.4 Constructors

A constructor initializes objects when created.

```
class Student
{
    public string Name;
    public int Age;

    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

Student s = new Student("Rahul", 20);
Console.WriteLine(s.Name);
```

7.5 Encapsulation – Properties

```
class Employee
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

7.6 Inheritance

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating..."); }
}

class Dog : Animal
```

```
{
    public void Bark() { Console.WriteLine("Barking..."); }
}

Dog d = new Dog();
d.Eat();  // Inherited
d.Bark(); // Own method
```

7.7 Polymorphism

Compile-Time (Method Overloading)

```
class MathOps
{
    public int Add(int a, int b) => a + b;
    public double Add(double a, double b) => a + b;
}
```

Run-Time (Method Overriding)

```
class Animal
{
    public virtual void Speak() { Console.WriteLine("Animal sound"); }
}

class Dog : Animal
{
    public override void Speak() { Console.WriteLine("Bark"); }
}
```

7.8 Abstraction with Interfaces & Abstract Classes

Interface:

```
interface IVehicle
{
    void Drive();
}
```

```
class Car : IVehicle
{
    public void Drive() => Console.WriteLine("Car is driving...");
}
```

Abstract Class:

```
abstract class Shape
{
    public abstract double Area();
}

class Circle : Shape
{
    public double Radius;
    public Circle(double r) { Radius = r; }
    public override double Area() => Math.PI * Radius * Radius;
}
```

7.9 Real-World Example – E-commerce System

- **Class:** `Product` (name, price).
- **Inheritance:** `Electronics : Product`, `Clothing : Product`.
- **Polymorphism:** `CalculateDiscount()` method works differently for each category.
- **Abstraction:** Payment processing via `IPaymentGateway`.

7.10 Exercises

1. Create a class Book with title, author, and price.
2. Implement inheritance between `Vehicle`, `Car`, and `Bike`.
3. Write a program demonstrating polymorphism using Shape, `Circle`, and `Rectangle`.

Chapter 8: Exception Handling & Debugging

8.1 What is Exception Handling?

Exceptions are runtime errors that disrupt program execution. C# provides **try-catch-finally** blocks to handle them gracefully.

8.2 Syntax

```
try
{
    // code that may cause error
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
finally
{
    Console.WriteLine("Execution finished.");
}
```

8.3 Common Exceptions

- `DivideByZeroException`
- `NullReferenceException`
- `IndexOutOfRangeException`
- `FileNotFoundException`

8.4 Throwing Exceptions

```
if (age < 0)
{
    throw new ArgumentException("Age cannot be negative");
}
```

8.5 Custom Exceptions

```
class InvalidAgeException : Exception
{
    public InvalidAgeException(string msg) : base(msg) { }
}

if (age < 18)
    throw new InvalidAgeException("Age must be 18 or above.");
```

8.6 Debugging Techniques in Visual Studio

- Breakpoints
- Step Into / Step Over
- Watch Window
- Immediate Window
- Call Stack

8.7 Real-World Example – Banking Transaction

```
try
{
    double balance = 5000;
    double withdraw = 6000;

    if (withdraw > balance)
        throw new Exception("Insufficient funds!");

    balance -= withdraw;
    Console.WriteLine("Remaining Balance: " + balance);
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

8.8 Exercises

1. Write a program to divide two numbers with exception handling.
2. Handle `IndexOutOfRangeException` while accessing an array.
3. Create a custom exception for invalid login attempts.

Chapter 9: Exercises, Mini Projects & Case Studies

9.1 Mini Project – Library Management System (Console App)

9.2 Practice Problems

9.3 Case Studies – ATM System, Student Management, E-commerce

Section 2 – ASP.NET Core & Web Development Basics

Chapter 1: Introduction to ASP.NET Core & MVC Architecture

1.1 What is ASP.NET Core?

ASP.NET Core is a **modern, cross-platform framework** developed by Microsoft for building **web applications, APIs, and microservices**. It runs on **Windows, Linux, and macOS**.

Key points:

- Open-source and modular
- High performance
- Cross-platform
- Built for cloud-based apps
- Supports MVC, Razor Pages, Blazor, Web APIs

1.2 Features of ASP.NET Core

1. **Cross-Platform:** Run on Windows, Linux, or macOS.
2. **High Performance:** Lightweight and optimized for modern web apps.
3. **Unified Framework:** Single framework for web, mobile, cloud, and APIs.
4. **Dependency Injection:** Built-in support for DI to decouple components.
5. **Middleware Pipeline:** Flexible request processing with custom middlewares.
6. **Razor Syntax:** Simplified HTML with C# code integration.
7. **Built-in Security:** Authentication, authorization, and data protection.

1.3 Differences Between ASP.NET Framework & ASP.NET Core

Feature	ASP.NET Framework	ASP.NET Core
Platform	Windows only	Cross-platform
Performance	Moderate	High
Dependency Injection	External library	Built-in
Modularity	Monolithic	Modular, lightweight
Hosting	IIS	IIS, Kestrel, Nginx

1.4 Understanding MVC Architecture

MVC (Model-View-Controller) is a **design pattern** that separates an application into three main components:

1. **Model** → Represents the application data and business logic.
2. **View** → User interface, displays data to the user.
3. **Controller** → Handles user requests, interacts with models, and returns views.

Diagram – MVC Flow:

```
User -> Controller -> Model -> Database
Controller -> View -> User
```

1.5 Benefits of Using MVC

- **Separation of Concerns:** Each layer handles a distinct responsibility.

- **Testability:** Easier to write unit tests for controllers and models.
- **Scalability:** Modular structure supports large applications.
- **Maintainability:** Easier to update and modify code.

1.6 Real-World Example – Simple Web App

Scenario: A **Product Catalog Website**

- Model → Product class (Name, Price, Description)
- View → Razor page displaying product list
- Controller → Handles requests like `/Products/List` and returns view

Controller Code Example:

```
public class ProductsController : Controller
{
    public IActionResult List()
    {
        List<Product> products = new List<Product>
        {
            new Product { Name="Laptop", Price=50000 },
            new Product { Name="Phone", Price=15000 }
        };
        return View(products);
    }
}
```

1.7 Exercises

1. Explain in your own words why MVC improves maintainability.
2. List at least 3 real-world applications where MVC is used.
3. Create a simple Controller returning a hardcoded string as a View.

Chapter 2: Project Structure in ASP.NET Core, Middleware & Pipeline

2.1 ASP.NET Core Project Structure Overview

When you create a new ASP.NET Core project, key folders and files include:

- **Program.cs** → Entry point of the application.
- **Startup.cs** → Configures services and middleware pipeline.
- **wwwroot** → Static files (CSS, JS, images).
- **Controllers** → Contains controller classes.
- **Views** → Razor views for UI.
- **Models** → Application data structures.
- **appsettings.json** → Configuration file (DB connections, app settings).

2.2 Program.cs & Startup.cs Explained

- **Program.cs** → Sets up the host and application entry.
- **Startup.cs** → Configures:
  - Services (Dependency Injection)
  - Middleware (request handling pipeline)

Example – Minimal Program.cs:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

2.3 Middleware – Concept & Usage

Middleware are **components that handle requests and responses** in the pipeline.

- Common middlewares: Authentication, Logging, Routing, Static Files.
- Order matters – request flows through each middleware sequentially.

Example – Custom Middleware:

```
app.Use(async (context, next) =>
{
    Console.WriteLine("Request incoming: " + context.Request.Path);
    await next();
    Console.WriteLine("Response outgoing: " + context.Response.StatusCode);
});
```

2.4 Request Processing Pipeline

1. Request enters the **server (Kestrel)**
2. Goes through **middlewares** (auth, logging, routing)
3. Hits **Controller/Endpoint**
4. Response is sent back to the client

2.5 Writing Custom Middleware

Step 1: Create a Middleware class

```
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;
    public LoggingMiddleware(RequestDelegate next) { _next = next; }

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine("Processing request...");
        await _next(context);
        Console.WriteLine("Processing response...");
    }
}
```

Step 2: Register in `Startup.cs`

```
app.UseMiddleware<LoggingMiddleware>();
```

2.6 Exercises

1. Draw a diagram showing how a request flows through middleware.

2. Implement middleware that logs every request URL and timestamp.
3. Explain the difference between built-in and custom middleware.

Chapter 3: Controllers & Action Methods

3.1 Introduction to Controllers

Controllers handle **HTTP requests** and return responses.

- Each controller typically represents a **feature/module**.

Example: ProductsController handles all product-related requests.

3.2 Creating a Controller

```
using Microsoft.AspNetCore.Mvc;

public class ProductsController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

3.3 Action Methods Explained

- Methods inside a controller that respond to requests.
- Can return:
    - **ViewResult** → renders a View
    - **JsonResult** → returns JSON data
    - **RedirectResult** → redirects to another action/page

Example – Returning JSON:

```
public IActionResult GetProducts()
{
    var products = new[] { "Laptop", "Phone" };
```

```
    return Json(products);
}
```

3.4 Routing in ASP.NET Core

- **Default Route:** `"{controller=Home}/{action=Index}/{id?}"`
- Maps URL segments to controllers and actions.

Example: `/Products/List` → `ProductsController.List()`

3.5 Attribute Routing

```
[Route("products")]
public class ProductsController : Controller
{
    [Route("list")]
    public IActionResult List() { ... }

    [Route("details/{id}")]
    public IActionResult Details(int id) { ... }
}
```

3.6 Exercises

1. Create a `CustomerController` with actions: `Index`, `Details`, `List`.
2. Return JSON data from an action method.
3. Implement attribute routing for all actions in `CustomerController`.

Chapter 4: Views, Razor Syntax & Layout Pages

4.1 Introduction to Views

Views are **UI templates** that render HTML content to the client. In ASP.NET Core MVC:

- Stored in the `Views` folder
- Typically `.cshtml` files
- Can be strongly-typed to use models

4.2 Razor Syntax Basics

Razor allows embedding **C# code within HTML** using the @ symbol.

Example:

```
<h1>Hello, @ViewBag.Name!</h1>

@for(int i = 1; i <= 5; i++)
{
    <p>Iteration: @i</p>
}
```

4.3 Layout Pages & Partial Views

- **Layout Pages:** Define common UI elements (header, footer, navigation) across all pages
- **Partial Views:** Reusable chunks of HTML rendered inside other views

Example – Layout Page: `_Layout.cshtml`

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewData["Title"] - My App</title>
</head>
<body>
    <header>My Header</header>
    <main>@RenderBody()</main>
    <footer>My Footer</footer>
</body>
</html>
```

Example – Using Partial View:

```
@Html.Partial("_Navigation")
```

4.4 Using Sections and RenderBody

- `@RenderBody()` → Main content placeholder in layout
- `@RenderSection("Scripts", required: false)` → Optional section for scripts

4.5 ViewData, ViewBag, and TempData

Feature	Description	Example
ViewData	Dictionary object, used to pass data from Controller to View	`ViewData["Message"] = "Hello";`
ViewBag	Dynamic wrapper around ViewData	`ViewBag.Message = "Hello";`
TempData	Preserves data for next request (Redirect scenarios)	`TempData["Status"] = "Saved";`

4.6 Strongly-Typed Views

```
// Model
public class Product { public string Name; public int Price; }

// Controller
public IActionResult List()
{
    List<Product> products = new List<Product>
    {
        new Product { Name="Laptop", Price=50000 },
        new Product { Name="Phone", Price=15000 }
    };
    return View(products);
}

// View (List.cshtml)
@model List<Product>
@foreach(var p in Model)
{
    <p>@p.Name - @p.Price</p>
```

```
}
```

4.7 Exercises

1. Create a layout page with header, footer, and navigation.
2. Pass data to a view using ViewBag and ViewData.
3. Implement a strongly-typed view for a `Student` model showing Name and Age.

Chapter 5: Models & ViewModels, Data Binding

5.1 Introduction to Models

Models represent **application data and business logic**.

Example:

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

5.2 Understanding ViewModels

- ViewModels are **custom classes** used to pass **specific data** to the view.
- Decouple domain models from UI requirements.

Example:

```
public class StudentViewModel
{
    public string Name { get; set; }
    public string Grade { get; set; }
}
```

5.3 Passing Data from Controller to View

```
public IActionResult Details()
{
    StudentViewModel student = new StudentViewModel { Name="Rahul", Grade="A" };
    return View(student);
}

<!-- View -->
<p>Name: @Model.Name</p>
<p>Grade: @Model.Grade</p>
```

5.4 Model Binding Basics

- Model binding maps **HTTP request data** (form values, query strings) to **controller action parameters**.

Example:

```
public IActionResult Register(string name, int age)
{
    return Content($"Name: {name}, Age: {age}");
}
```

5.5 Validation using Data Annotations

- `[Required]` → Field must be filled
- `[StringLength(50)]` → Max length
- `[Range(1, 100)]` → Valid range

Example – Model with Validation:

```
public class Student
{
    [Required]
    public string Name { get; set; }

    [Range(1, 100)]
    public int Age { get; set; }
```

```
}
```

5.6 Exercises

1. Create a `Product` model and pass a list of products to a view.
2. Implement a `StudentViewModel` with Name and Grade, and display it in a view.
3. Apply data annotations to validate student input.

Chapter 6: Form Handling & Validation

6.1 Creating HTML Forms in Razor Views

```
<form asp-action="Register" method="post">
    <label>Name:</label>
    <input type="text" name="Name" />
    <label>Age:</label>
    <input type="number" name="Age" />
    <button type="submit">Submit</button>
</form>
```

6.2 Binding Form Data to Models

```
[HttpPost]
public IActionResult Register(Student student)
{
    if(ModelState.IsValid)
        return Content($"Student {student.Name} registered successfully!");
    else
        return View(student);
}
```

6.3 Server-Side Validation

- Use **ModelState.IsValid** to check validation attributes.
- Return view with errors if invalid.

```
<span asp-validation-for="Name"></span>
<span asp-validation-for="Age"></span>
```

6.4 Client-Side Validation

- Razor automatically generates **jQuery validation** scripts if included.
- Prevents submission of invalid data before reaching the server.

6.5 Validation Summary & Validation Messages

```
<div asp-validation-summary="All"></div>
```

Displays all validation errors at the top of the form.

6.6 Real-World Example – Registration Form

1. Model: `Student` with Name, Age, Email
2. Controller: Accepts POST requests, validates model
3. View: Displays form with validation messages

```
[HttpPost]
public IActionResult Register(Student student)
{
    if(ModelState.IsValid)
        TempData["Success"] = "Student registered!";
    return View(student);
}
```

6.7 Exercises

1. Create a registration form for a product with Name, Price, and Quantity.
2. Implement client-side and server-side validation for the form.
3. Display validation errors using `asp-validation-summary`.

Chapter 7: Dependency Injection (DI) in ASP.NET Core

7.1 What is Dependency Injection?

Dependency Injection is a **design pattern** used to achieve **loose coupling** between classes and their dependencies.

- The **service** is provided rather than the class creating it.
- Promotes testability and maintainability.

7.2 Built-in DI in ASP.NET Core

- ASP.NET Core has a **built-in DI container**.
- Register services in `Program.cs` or `Startup.cs`
- Inject services via **constructor injection**

7.3 Registering Services

Service Lifetime	Description	Example
Transient	Created each time requested	`services.AddTransient<IService, Service>()`
Scoped	Created once per request	`services.AddScoped<IService, Service>()`
Singleton	Created once for application lifetime	`services.AddSingleton<IService, Service>()`

7.4 Injecting Services into Controllers

```
public interface IGreetingService
{
    string Greet(string name);
}

public class GreetingService : IGreetingService
{
    public string Greet(string name) => $"Hello, {name}!";
}
```

```
// Program.cs
builder.Services.AddTransient<IGreetingService, GreetingService>();

// Controller
public class HomeController : Controller
{
    private readonly IGreetingService _greetingService;

    public HomeController(IGreetingService greetingService)
    {
        _greetingService = greetingService;
    }

    public IActionResult Index()
    {
        string message = _greetingService.Greet("Rahul");
        return Content(message);
    }
}
```

7.5 Real-World Example – Logging Service

- Create a service that logs requests and inject it into multiple controllers
- Ensures consistent logging across the application

7.6 Exercises

1. Implement a `NotificationService` and inject it into two controllers.
2. Explain the difference between Transient, Scoped, and Singleton services.
3. Create a simple DI example with multiple services interacting.

Chapter 8: Introduction to Entity Framework Core (EF Core)

8.1 What is EF Core?

Entity Framework Core is a **modern Object-Relational Mapper (ORM)** for .NET:

- Maps **C# classes to database tables**
- Supports LINQ queries
- Handles CRUD operations automatically

8.2 Benefits of Using EF Core

- Reduces boilerplate SQL code
- Supports **code-first and database-first approaches**
- Works with multiple databases (SQL Server, SQLite, MySQL)
- Built-in migrations for schema changes

8.3 DbContext & DbSet Explained

- **DbContext:** Main class for database interaction
- **DbSet<T>:** Represents a table

```
public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
{ }
}
```

8.4 Configuring Connection Strings

`appsettings.json` example:

```
"ConnectionStrings": {
  "DefaultConnection": "Server=.;Database=MyAppDb;Trusted_Connection=True;"
}

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection
")));
```

8.5 Code-First Approach Overview

- Define models in C# classes
- Use migrations to create database

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

8.6 Database-First Approach Overview

- Generate models from an existing database

```
dotnet ef dbcontext scaffold "ConnectionString"
Microsoft.EntityFrameworkCore.SqlServer -o Models
```

8.7 Exercises

1. Create a `Product` model and set up EF Core DbContext.
2. Generate database using code-first migrations.
3. Scaffold models from an existing database (database-first).

Chapter 9: Database Operations with EF Core

9.1 CRUD Operations – Create, Read, Update, Delete

Create:

```
var product = new Product { Name = "Laptop", Price = 50000 };
_dbContext.Products.Add(product);
_dbContext.SaveChanges();
```

Read:

```
var products = _dbContext.Products.ToList();
```

Update:

```
var product = _dbContext.Products.First();
product.Price = 55000;
_dbContext.SaveChanges();
```

Delete:

```
var product = _dbContext.Products.First();
_dbContext.Products.Remove(product);
_dbContext.SaveChanges();
```

9.2 LINQ Queries

- Filter, sort, and select data using LINQ

```
var expensiveProducts = _dbContext.Products
                        .Where(p => p.Price > 30000)
                        .OrderByDescending(p => p.Price)
                        .ToList();
```

9.3 Using Migrations

- Add new columns or tables using `Add-Migration`
- Update database using `Update-Database`
- Keeps schema in sync with models

9.4 Relationships (One-to-One, One-to-Many, Many-to-Many)

One-to-Many Example:

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

public class Product
{
```

```
    public int Id { get; set; }
    public string Name { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}
```

9.5 Real-World Example – Employee Management App

- Models: Employee, Department
- CRUD operations for employees
- LINQ queries to filter employees by department
- Relationships mapped using EF Core

9.6 Exercises

1. Create a `Student` model and perform CRUD operations.
2. Implement a one-to-many relationship between `Course` and `Student`.
3. Write LINQ queries to fetch top 5 students by marks.

Chapter 10: Final Mini Project

10.1 Project Overview

Project: Simple Product Catalog Web Application

- Features: List products, add, edit, delete products, filter products by category

10.2 Designing Models, Views, and Controllers

- **Models:** Product, Category
- **Controllers:** ProductsController, CategoriesController
- **Views:** List, Create, Edit, Delete

10.3 Implementing CRUD Operations

- Create form using Razor pages
- Bind form data to models
- Save data to SQL Server using EF Core

10.4 Validation & Exception Handling

- Validate form fields using **Data Annotations**
- Handle errors using **try-catch** blocks in controllers

10.5 Deployment Overview

- Host on **IIS, Azure, or Docker**
- Configure **appsettings.json** for production DB
- Ensure proper logging and exception handling

10.6 Exercises & Enhancements

1. Add search functionality to filter products by name
2. Add authentication and role-based access
3. Implement AJAX calls for dynamic updates

Section 3 – Angular Basics

Chapter 1: Introduction to Angular

1.1 What is Angular?

Angular is a **modern, open-source frontend framework** developed by Google for building **dynamic single-page applications (SPAs)**.

- Written in **TypeScript**
- Component-based architecture
- Uses **two-way data binding** to synchronize UI and data

1.2 History and Evolution of Angular

1. **AngularJS (1.x)** – Original JavaScript-based framework
2. **Angular 2+** – Complete rewrite with TypeScript
3. **Angular 4, 5, 6...** – Continuous improvements, modular and faster
4. **Angular 15/16+** – Latest versions with better performance and CLI support

1.3 Angular vs AngularJS

Feature	AngularJS	Angular (2+)
Language	JavaScript	TypeScript
Architecture	MVC	Component-based
Performance	Slower for large apps	High performance
Mobile Support	Limited	Full support
Dependency Injection	Basic	Advanced

1.4 Features of Angular

- **Component-based architecture**
- **Two-way data binding**
- **Dependency Injection (DI)**
- **Routing & Navigation**
- **Directives (Structural & Attribute)**
- **Services for reusable logic**

- **HTTP Client for API calls**

1.5 Benefits of Using Angular

- Reduces boilerplate code
- Improves maintainability and scalability
- Easy integration with backend APIs
- Strong community support
- Works for desktop, mobile, and web apps

1.6 Real-World Examples of Angular Applications

- Gmail
- Microsoft Office 365
- Forbes website
- Upwork dashboard

1.7 Exercises

1. Explain in your own words why Angular is preferred over AngularJS.
2. List 3 real-world apps built using Angular.
3. Write a paragraph describing the benefits of a component-based architecture.

Chapter 2: Setting Up Angular Development Environment

2.1 Installing Node.js and npm

Angular requires **Node.js** and **npm (Node Package Manager)**.

- Download Node.js from https://nodejs.org
- Verify installation:

```
node -v
npm -v
```

2.2 Installing Angular CLI

Angular CLI helps **generate projects, components, services, and modules**.

```
npm install -g @angular/cli
```

Verify installation:

```
ng version
```

2.3 Creating a New Angular Project

```
ng new my-angular-app
cd my-angular-app
ng serve
```

- The app runs at http://localhost:4200
- CLI generates **folders and files automatically**

2.4 Project Structure Overview

Folder/File	Description
src/app	Main application folder
app.module.ts	Root module
app.component.ts	Root component logic
app.component.html	Root component view
assets/	Images, fonts, other static files
environments/	Configuration for dev and prod

2.5 Running Angular Application

- Start development server: `ng serve`
- Open browser: http://localhost:4200

- Hot reload available – saves changes automatically

2.6 Exercises

1. Install Node.js and Angular CLI on your system.
2. Create a new Angular project and run it on localhost.
3. Explore the generated folder structure and list main files and folders.

Chapter 3: Angular Components

3.1 Introduction to Components

Components are the **building blocks of Angular applications**.

- Encapsulate logic, template, and styles
- Represent a **view/UI section**

3.2 Creating Components Using CLI

```
ng generate component products
```

- Creates `products.component.ts`, `.html`, `.css`, and `.spec.ts` files
- Updates `app.module.ts` automatically

3.3 Component Decorator and Metadata

```
@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent {}
```

- **selector** → HTML tag for the component

- **templateUrl** → HTML template file
- **styleUrls** → CSS file(s)

3.4 Template, Style, and Selector

- Template: HTML content displayed in the browser
- Styles: CSS applied to the component
- Selector: Tag used in parent component or root template

```
<app-products></app-products>
```

3.5 Lifecycle Hooks of Components

- `ngOnInit()` – Called after component initialization
- `ngOnChanges()` – Called when input properties change
- `ngOnDestroy()` – Cleanup before component destruction

3.6 Nested Components

- Parent component can include child components using **selector tags**
- Promotes **modularity and reusability**

3.7 Exercises

1. Create a `Student` component with template displaying student name and age.
2. Implement `ngOnInit` to initialize a list of students.
3. Create a parent component and nest the `Student` component inside it.

Chapter 4: Data Binding in Angular

4.1 Introduction to Data Binding

Data binding in Angular is a mechanism to **synchronize data between the component and the view**. It ensures the **UI reflects changes in data** and vice versa.

4.2 Types of Data Binding

4.2.1 Interpolation

- Display component data in the template using `{{ }}`

Example:

```
// Component
export class AppComponent {
  title: string = "Angular Basics";
}

<h1>{{ title }}</h1>
```

4.2.2 Property Binding

- Bind **HTML element properties** to component data using `[property]`

Example:

```
<input [value]="title" />
```

4.2.3 Event Binding

- Bind **events** (click, change, submit) to component methods using `(event)`

Example:

```
<button (click)="showMessage()">Click Me</button>
```

```
showMessage() {
  alert("Button Clicked!");
}
```

4.2.4 Two-Way Binding

- Bind data in both directions using `[(ngModel)]`
- Requires importing `FormsModule` in `app.module.ts`

Example:

```
<input [(ngModel)]="title" />
<p>{{ title }}</p>
```

4.3 Real-World Examples of Data Binding

- Search box updating list dynamically
- Live preview of form inputs
- Updating a shopping cart total automatically

4.4 Exercises

1. Create an input field bound to a component variable using interpolation.
2. Implement a button click event to update a message on the screen.
3. Create a two-way binding form input and display its value in real-time.

Chapter 5: Angular Directives

5.1 Introduction to Directives

Directives are **special instructions in templates** that **alter the DOM behavior or structure**.

Types:

- **Structural Directives:** Change DOM layout (`*ngIf`, `*ngFor`)
- **Attribute Directives:** Change element appearance (`[ngStyle]`, `[ngClass]`)

5.2 Structural Directives

*5.2.1 *ngIf*

- Conditionally includes or removes an element

Example:

```
<p *ngIf="isVisible">This paragraph is visible.</p>
```

*5.2.2 *ngFor*

- Loop through arrays to render elements

Example:

```
<ul>
  <li *ngFor="let student of students">{{ student.name }}</li>
</ul>
```

5.3 Attribute Directives

5.3.1 [ngStyle]

- Apply dynamic styles

Example:

```
<p [ngStyle]="{'color': isRed ? 'red' : 'blue'}">Colored Text</p>
```

5.3.2 [ngClass]

- Apply dynamic CSS classes

Example:

```
<p [ngClass]="{ 'highlight': isHighlighted }">Class Example</p>
```

5.4 Custom Directives

- Create reusable directives to **modify DOM behavior**

Example – Change text color:

```
@Directive({ selector: '[appHighlight]' })
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.color = 'blue';
  }
}

<p appHighlight>This text is highlighted.</p>
```

5.5 Exercises

1. Use `*ngIf` to toggle visibility of a paragraph.
2. Use `*ngFor` to display a list of products.
3. Create a custom directive to change the background color of a div.

Chapter 6: Angular Services and Dependency Injection

6.1 What is a Service in Angular?

- Services are **classes containing reusable logic**
- Can be used across multiple components
- Promotes **separation of concerns**

6.2 Creating and Using Services

Step 1 – Generate a Service:

```
ng generate service student
```

Step 2 – Service Logic:

```
@Injectable({
  providedIn: 'root'
})
export class StudentService {
  students = ['Rahul', 'Anita', 'Vikram'];

  getStudents() {
    return this.students;
  }
}
```

Step 3 – Inject Service into Component:

```
export class AppComponent {
  students: string[];

  constructor(private studentService: StudentService) {
    this.students = this.studentService.getStudents();
  }
}
```

6.3 Dependency Injection in Angular

- Angular automatically **injects service instances** where required
- Service registered in `root` is **singleton** by default

6.4 Singleton Services

- Ensures only **one instance of the service** exists across the app
- Useful for **shared data or global state management**

6.5 Exercises

1. Create a `ProductService` to return a list of products and display in a component.
2. Create a singleton service to manage user authentication state.
3. Inject a service into multiple components and use shared data.

Chapter 7: Angular Routing and Navigation

7.1 Introduction to Routing

Routing in Angular allows **navigation between different views/components** without reloading the page.

- Part of the **Angular Router module**
- Enables **single-page application (SPA)** behavior

7.2 Configuring Routes in Angular

Step 1 – Import RouterModule in `app.module.ts`

```
import { RouterModule, Routes } from '@angular/router';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { ProductsComponent } from './products/products.component';
import { HomeComponent } from './home/home.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'products', component: ProductsComponent }
];

@NgModule({
  declarations: [AppComponent, ProductsComponent, HomeComponent],
  imports: [BrowserModule, RouterModule.forRoot(routes)],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

7.3 RouterLink and RouterOutlet

- `<router-outlet>` → Placeholder where routed components render
- `[routerLink]` → Navigation links

```
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/products">Products</a>
</nav>

<router-outlet></router-outlet>
```

7.4 Route Parameters

- Pass data through routes using `:id`

```
{ path: 'products/:id', component: ProductDetailComponent }
```

- Access in component:

```
import { ActivatedRoute } from '@angular/router';

constructor(private route: ActivatedRoute) {}
ngOnInit() {
  let productId = this.route.snapshot.params['id'];
}
```

7.5 Nested Routes

- Routes can have **child routes**

```
{ path: 'products', component: ProductsComponent,
  children: [
    { path: 'details/:id', component: ProductDetailComponent }
  ]
}
```

7.6 Exercises

1. Create a home component and products component with navigation using RouterLink.
2. Implement route parameters to display product details.
3. Add a nested route for product reviews inside the product details component.

Chapter 8: Angular Forms

8.1 Introduction to Forms in Angular

Angular provides **two approaches** to handle forms:

1. **Template-Driven Forms** – Simple, declarative approach
2. **Reactive Forms** – Programmatic, more scalable

8.2 Template-Driven Forms

Steps:

1. Import `FormsModule` in `app.module.ts`
2. Bind form controls with `[(ngModel)]`

```
<form #f="ngForm" (ngSubmit)="submitForm(f)">
  <label>Name:</label>
  <input type="text" name="name" ngModel required>
  <button type="submit">Submit</button>
</form>

submitForm(form: any) {
  console.log(form.value);
}
```

8.3 Reactive Forms

Steps:

1. Import `ReactiveFormsModule` in `app.module.ts`
2. Create `FormGroup` and `FormControl`

```
import { FormGroup, FormControl } from '@angular/forms';

this.productForm = new FormGroup({
  name: new FormControl(''),
  price: new FormControl('')
});

submitForm() {
  console.log(this.productForm.value);
}

<form [formGroup]="productForm" (ngSubmit)="submitForm()">
  <input formControlName="name">
  <input formControlName="price">
  <button type="submit">Submit</button>
</form>
```

8.4 Form Validation (Built-in Validators & Custom Validators)

- Validators: `required`, `minLength`, `maxLength`, `pattern`
- Custom Validator Example:

```
function priceValidator(control: FormControl) {
  return control.value > 0 ? null : { invalidPrice: true };
}
```

8.5 Exercises

1. Create a template-driven form for student registration.
2. Create a reactive form for product input with validators.
3. Implement a custom validator to check if price is positive.

Chapter 9: Angular HTTP Client and APIs

9.1 Introduction to HTTP Client

Angular HTTPClient is used to **communicate with backend APIs**.

- Supports GET, POST, PUT, DELETE requests
- Works with **Observables** (RxJS)

9.2 Making GET, POST, PUT, DELETE Requests

GET Request Example:

```
this.http.get('https://api.example.com/products').subscribe(data => {
  this.products = data;
});
```

POST Request Example:

```
this.http.post('https://api.example.com/products', product).subscribe();
```

PUT Request Example:

```
this.http.put('https://api.example.com/products/1', product).subscribe();
```

DELETE Request Example:

```
this.http.delete('https://api.example.com/products/1').subscribe();
```

9.3 Observables and RxJS Basics

- Observables provide **asynchronous data handling**
- Methods: `subscribe()`, `map()`, `filter()`

9.4 Handling HTTP Responses and Errors

```
this.http.get('https://api.example.com/products').subscribe({
  next: data => this.products = data,
  error: err => console.error('Error', err)
});
```

9.5 Real-World Example – Connecting to a REST API

- Fetch products from backend (ASP.NET Core API)
- Display list in Angular component
- Add, update, delete products via API calls

9.6 Exercises

1. Make a GET request to fetch data and display in a component.
2. Implement POST request to add new products.
3. Handle errors and display error messages to users.

Chapter 10: Final Mini Project

10.1 Project Overview – Angular CRUD App

Project Features:

- Product listing
- Add/Edit/Delete product
- Form validation
- API integration with ASP.NET Core backend
- Routing for product details

10.2 Creating Components and Services

- Components: `ProductList`, `ProductDetail`, `ProductForm`
- Services: `ProductService` for API calls

10.3 Implementing Routing

- Configure routes for each component
- Use `routerLink` and `router-outlet`

10.4 Form Handling and Validation

- Reactive forms for product creation/editing
- Validators for required fields, number ranges, and text patterns

10.5 Integrating with Backend API (ASP.NET Core)

- Use Angular HTTPClient to **connect to ASP.NET Core API**
- Perform CRUD operations using REST endpoints

10.6 Exercises & Enhancements

1. Add search and filter functionality to the product list.
2. Implement pagination for large datasets.
3. Add authentication to restrict editing and deleting of products.
4. Add notifications for success or error messages.

Chapter 1: Introduction to Full Stack Integration

1.1 What is Full Stack Development?

Full Stack Development involves **working on both the frontend and backend** of a web application, along with database management.

- **Frontend:** User Interface (UI) / Client-side logic (Angular in our syllabus)
- **Backend:** Server-side logic, APIs, and data processing (ASP.NET Core)
- **Database:** Storage and retrieval of data (SQL Server or other RDBMS)

Key Responsibilities of a Full Stack Developer:

1. Design user interfaces using Angular
2. Create server-side APIs using ASP.NET Core
3. Connect frontend and backend for data exchange
4. Implement authentication, authorization, and security
5. Optimize performance and handle deployment

1.2 Overview of Frontend (Angular) and Backend (ASP.NET Core) Integration

How Angular and ASP.NET Core Work Together:

- Angular handles the **presentation layer** – displaying data, forms, and UI elements
- ASP.NET Core provides **RESTful APIs** for data management
- Communication occurs via **HTTP requests (GET, POST, PUT, DELETE)**
- JSON is used for **data exchange** between frontend and backend

Example Flow:

1. User submits a form in Angular
2. Angular sends a POST request to ASP.NET Core API
3. API processes data and saves it to the database
4. API returns a response (success/failure)
5. Angular updates the UI based on the response

1.3 Benefits of Full Stack Applications

1. **End-to-End Development:** Build complete applications without depending on multiple developers

2. **Reusability:** Services and components can be reused across projects
3. **Scalability:** Easier to scale apps with component-based frontend and modular backend
4. **Rapid Prototyping:** Can quickly build MVPs (Minimum Viable Products)
5. **Better Collaboration:** Clear separation of frontend and backend roles, yet easy integration

1.4 Architecture of Full Stack Apps (Client-Server-Database)

Three-Tier Architecture:

1. **Client Layer (Angular):** UI, forms, validations, data display
2. **Server Layer (ASP.NET Core API):** Business logic, authentication, API endpoints
3. **Database Layer (SQL Server):** Persistent data storage

Diagram Example:

```
[ Angular Frontend ]
        |
        | HTTP Requests (JSON)
        v
[ ASP.NET Core API / Backend ]
        |
        | SQL Queries / ORM (EF Core)
        v
[ Database (SQL Server) ]
```

Key Points:

- Client interacts only with API, not the database directly
- Backend handles all business logic and data processing
- Database stores structured data

1.5 Exercises

1. Draw a diagram representing the flow of data in a full stack Angular + ASP.NET Core application.
2. List three advantages of using a full stack approach over frontend-only or backend-only development.
3. Explain in your own words how Angular communicates with ASP.NET Core APIs.

1.6 Real-World Full Stack Examples

- **E-Commerce Website:** Product listing (Angular), Order Management API (ASP.NET Core), SQL database
- **Employee Management System:** Employee forms (Angular), Employee API (ASP.NET Core), HR database
- **Social Media App:** Post feed (Angular), API for posts/comments (ASP.NET Core), User database

Chapter 2: Setting Up Full Stack Environment

2.1 ASP.NET Core API Project Setup

Step 1 – Create a New API Project

```
dotnet new webapi -n FullStackAPI
cd FullStackAPI
dotnet restore
dotnet run
```

- Creates a project named `FullStackAPI`
- Runs API at https://localhost:5001 or http://localhost:5000

Step 2 – Project Structure Overview

Folder/File	Description
`Controllers/`	API controller files
`Models/`	Data models (C# classes)
`Program.cs`	Main entry point, DI setup
`appsettings.json`	Database connection and configuration

2.2 Angular Frontend Project Setup

Step 1 – Create Angular Project

```
ng new FullStackFrontend
cd FullStackFrontend
```

```
ng serve
```

- Runs Angular app at http://localhost:4200
- CLI generates folders: src/app, assets, environments

Step 2 – Project Structure Overview

Folder/File	Description
app/	Contains components, services, modules
app.module.ts	Root module
app.component.ts	Root component
app.component.html	Root template

2.3 Connecting Angular Frontend with ASP.NET Core API

Step 1 – Create Angular Service

```
ng generate service services/product
```

Step 2 – HTTP GET Example

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Product } from '../models/product.model';

@Injectable({ providedIn: 'root' })
export class ProductService {
  private apiUrl = 'https://localhost:5001/api/products';

  constructor(private http: HttpClient) {}

  getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.apiUrl);
  }
}
```

Step 3 – Use Service in Component

```
import { Component, OnInit } from '@angular/core';
import { ProductService } from './services/product.service';
import { Product } from './models/product.model';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html'
})
export class ProductListComponent implements OnInit {
  products: Product[] = [];

  constructor(private productService: ProductService) {}

  ngOnInit() {
    this.productService.getProducts().subscribe(data => this.products = data);
  }
}
```

Step 4 – Display Data in Template

```
<ul>
  <li *ngFor="let product of products">
    {{ product.name }} - {{ product.price }}
  </li>
</ul>
```

2.4 CORS Configuration

Problem: Angular frontend and ASP.NET Core backend run on different ports, which triggers **Cross-Origin Resource Sharing (CORS)** errors.

Solution: Add CORS in `Program.cs`

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAngular",
        policy => policy.WithOrigins("http://localhost:4200")
                        .AllowAnyMethod()
                        .AllowAnyHeader());
```

```
});

builder.Services.AddControllers();
var app = builder.Build();

app.UseCors("AllowAngular");
app.UseAuthorization();
app.MapControllers();
app.Run();
```

- `WithOrigins("http://localhost:4200")` → Allows requests from Angular frontend

2.5 Exercises

1. Create an ASP.NET Core API project and run it locally.
2. Create an Angular project and run it on localhost.
3. Configure Angular to fetch data from ASP.NET Core API using a service.
4. Enable CORS in ASP.NET Core to allow Angular requests.
5. Create a simple product model and display data in Angular component.

2.6 Real-World Example

- **Scenario:** Employee Management System
  - Backend API: `/api/employees`
  - Frontend Angular Service: `EmployeeService`
  - Component: `EmployeeListComponent`
  - CORS configuration ensures Angular can fetch data without security errors

Chapter 3: Advanced Angular Concepts for Full Stack

3.1 Lazy Loading Modules

Definition:
Lazy loading allows Angular modules to be **loaded on demand**, improving **application startup performance**.

Example Scenario:

- An e-commerce app may load `AdminModule` only when an admin logs in.

Step 1 – Create a Module

```
ng generate module admin --route admin --module app.module
```

- CLI automatically configures lazy loading in `app-routing.module.ts`

Step 2 – Routing Configuration

```
const routes: Routes = [
  { path: 'admin', loadChildren: () => import('./admin/admin.module').then(m =>
m.AdminModule) }
];
```

Benefits:

- Faster initial load
- Modular architecture
- Reduced bundle size

Exercise:

1. Create a `ReportsModule` and configure lazy loading.
2. Test by navigating to `/reports` and observe network load.

3.2 Route Guards (AuthGuard, CanActivate)

Definition:
Route Guards **protect routes from unauthorized access**.

Types of Route Guards:

- `CanActivate` – Prevent navigation to a route

- `CanActivateChild` – Protect child routes
- `CanDeactivate` – Warn before leaving a route
- `Resolve` – Pre-fetch data before route activation

Example – AuthGuard

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';

@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  constructor(private router: Router) {}

  canActivate(): boolean {
    const isLoggedIn = !!localStorage.getItem('token');
    if (!isLoggedIn) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }
}
```

Apply Guard in Routing:

```
{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard] }
```

Exercise:

1. Implement an `AuthGuard` for admin routes.
2. Test unauthorized access redirection to login page.

3.3 Angular Interceptors for HTTP Requests

Definition:
Interceptors allow you to **intercept HTTP requests and responses** globally.

Use Cases:

- Add JWT token to headers
- Log requests/responses

- Handle global error responses

Example – JWT Interceptor:

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';

@Injectable()
export class JwtInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const token = localStorage.getItem('token');
    if (token) {
      req = req.clone({ setHeaders: { Authorization: `Bearer ${token}` } });
    }
    return next.handle(req);
  }
}
```

Register Interceptor in AppModule:

```
providers: [{ provide: HTTP_INTERCEPTORS, useClass: JwtInterceptor, multi:
true }]
```

Exercise:

1. Implement an interceptor to attach JWT token to all API requests.
2. Add a global error interceptor to handle API errors.

3.4 State Management Basics (BehaviorSubject, Services)

Definition:
State management allows **sharing data between components** without direct parent-child relationships.

BehaviorSubject Example:

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class AuthService {
  private loggedIn = new BehaviorSubject<boolean>(false);
```

```
  currentStatus = this.loggedIn.asObservable();

  login() { this.loggedIn.next(true); }
  logout() { this.loggedIn.next(false); }
}
```

Using in Component:

```
this.authService.currentStatus.subscribe(status => {
  this.isLoggedIn = status;
});
```

Exercise:

1. Create a `CartService` using `BehaviorSubject` to share cart items across components.
2. Implement add/remove functionality and display total items in header component.

3.5 Exercises Summary

1. Configure lazy loading for a `ReportsModule` and verify network requests.
2. Implement `AuthGuard` to protect admin routes.
3. Create an HTTP interceptor to attach JWT token to all requests.
4. Implement `BehaviorSubject` to manage global login state.
5. Combine guard and interceptor to protect secure API calls.

Chapter 4: Advanced ASP.NET Core Concepts

4.1 Repository Pattern

Definition:
The Repository Pattern provides a **layer of abstraction** over data access logic.

- Makes the application **more maintainable and testable**
- Decouples business logic from database logic

Example:

Step 1 – Create Interface:

```
public interface IProductRepository
{
    IEnumerable<Product> GetAll();
    Product GetById(int id);
    void Add(Product product);
    void Update(Product product);
    void Delete(int id);
}
```

Step 2 – Implement Repository:

```
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _context;
    public ProductRepository(AppDbContext context)
    {
        _context = context;
    }

    public IEnumerable<Product> GetAll() => _context.Products.ToList();

    public Product GetById(int id) => _context.Products.Find(id);

    public void Add(Product product)
    {
        _context.Products.Add(product);
        _context.SaveChanges();
    }

    public void Update(Product product)
    {
        _context.Products.Update(product);
        _context.SaveChanges();
    }

    public void Delete(int id)
    {
        var product = _context.Products.Find(id);
        if (product != null)
```

```
        {
            _context.Products.Remove(product);
            _context.SaveChanges();
        }
    }
}
```

Benefits:

- Centralized data access logic
- Easier unit testing
- Clear separation of concerns

4.2 Unit of Work Pattern

Definition:
The Unit of Work Pattern **manages multiple repository operations in a single transaction**.

- Ensures **data consistency**
- Useful when multiple tables are updated in one business operation

Example:

```
public interface IUnitOfWork : IDisposable
{
    IProductRepository Products { get; }
    ICategoryRepository Categories { get; }
    void Complete();
}

public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;
    public IProductRepository Products { get; private set; }
    public ICategoryRepository Categories { get; private set; }

    public UnitOfWork(AppDbContext context)
    {
        _context = context;
        Products = new ProductRepository(_context);
        Categories = new CategoryRepository(_context);
    }
```

```
    public void Complete() => _context.SaveChanges();

    public void Dispose() => _context.Dispose();
}
```

Exercise:

1. Implement `UnitOfWork` for Employee and Department repositories.
2. Update multiple entities in a single transaction.

4.3 Dependency Injection in Advanced Scenarios

Definition:
Dependency Injection (DI) allows **injecting dependencies into classes** rather than creating them manually.

Example – Constructor Injection:

```
public class ProductService
{
    private readonly IUnitOfWork _unitOfWork;
    public ProductService(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    public IEnumerable<Product> GetProducts() => _unitOfWork.Products.GetAll();
}
```

Register Services in `Program.cs`:

```
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
builder.Services.AddScoped<ProductService>();
```

Benefits:

- Decouples classes from concrete implementations
- Promotes testability and modularity

4.4 Middleware & Exception Handling

Middleware Definition:

Middleware is a **component in the HTTP request pipeline** that can process requests and responses.

Example – Logging Middleware:

```
public class RequestLoggingMiddleware
{
    private readonly RequestDelegate _next;
    public RequestLoggingMiddleware(RequestDelegate next) => _next = next;

    public async Task Invoke(HttpContext context)
    {
        Console.WriteLine($"Request Path: {context.Request.Path}");
        await _next(context);
    }
}

// Register in Program.cs
app.UseMiddleware<RequestLoggingMiddleware>();
```

Global Exception Handling:

```
app.UseExceptionHandler(appError =>
{
    appError.Run(async context =>
    {
        context.Response.ContentType = "application/json";
        var contextFeature = context.Features.Get<IExceptionHandlerFeature>();
        if (contextFeature != null)
        {
            await context.Response.WriteAsync(new
            { StatusCode = context.Response.StatusCode, Message =
contextFeature.Error.Message }.ToString());
        }
    });
});
```

Exercise:

1. Create middleware to log request time.
2. Implement global exception handling to return JSON error messages.

4.5 Logging and Monitoring

ASP.NET Core Logging Providers:

- Console
- Debug
- File
- Third-party (Serilog, NLog)

Example – Console Logging:

```
public class ProductService
{
    private readonly ILogger<ProductService> _logger;
    public ProductService(ILogger<ProductService> logger)
    {
        _logger = logger;
    }

    public void AddProduct(Product product)
    {
        _logger.LogInformation("Adding product {ProductName}", product.Name);
        // Add product logic
    }
}
```

Exercise:

1. Implement console logging for API requests.
2. Configure Serilog to log API errors to a file.

4.6 Exercises Summary

1. Implement Repository and UnitOfWork patterns for multiple models.
2. Configure DI for repositories and services.
3. Create middleware for logging requests and handle exceptions globally.
4. Add logging to track important actions in APIs.

Chapter 5: Authentication & Authorization

5.1 Introduction to Authentication and Authorization

Definition:

- **Authentication:** Verifies the identity of a user (login)
- **Authorization:** Determines what resources a user can access (roles/permissions)

Key Concepts:

- Users must log in with credentials
- Secure APIs require a valid token for access
- Role-based access ensures only authorized users can perform certain actions

Example:

- Admin can add/edit/delete products
- Regular users can only view products

5.2 JWT (JSON Web Token) Authentication

Definition:
JWT is a **compact token format** used to securely transmit user identity and claims between client and server.

Steps to Implement JWT in ASP.NET Core:

Step 1 – Install NuGet Package

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Step 2 – Configure JWT in `Program.cs`

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "MyApp",
        ValidAudience = "MyApp",
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("SuperSecretKey123"))
    };
});
```

Step 3 – Generate Token on Login

```
var claims = new[]
{
    new Claim(ClaimTypes.Name, user.Username),
    new Claim(ClaimTypes.Role, user.Role)
};

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("SuperSecretKey123"));
var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(
    issuer: "MyApp",
    audience: "MyApp",
    claims: claims,
    expires: DateTime.Now.AddHours(2),
    signingCredentials: creds
);

return Ok(new { token = new JwtSecurityTokenHandler().WriteToken(token) });
```

Step 4 – Secure API Endpoints

```
[Authorize]
[HttpGet("products")]
public IActionResult GetProducts() { ... }
```

```
[Authorize(Roles = "Admin")]
[HttpPost("products")]
public IActionResult AddProduct(Product product) { ... }
```

5.3 Role-Based Authorization

Definition:

- Restricts access based on **user roles**

Implementation:

- Define roles in JWT token
- Use `[Authorize(Roles = "RoleName")]` on API methods

Example:

```
[Authorize(Roles = "Admin,Manager")]
[HttpPut("products/{id}")]
public IActionResult UpdateProduct(int id, Product product) { ... }
```

5.4 Token Validation in Angular

Step 1 – Store Token on Login

```
localStorage.setItem('token', response.token);
```

Step 2 – Attach Token Using Interceptor

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  const token = localStorage.getItem('token');
  if (token) {
    req = req.clone({ setHeaders: { Authorization: `Bearer ${token}` } });
  }
  return next.handle(req);
}
```

Step 3 – Guard Routes Based on Role

```
canActivate(): boolean {
  const token = localStorage.getItem('token');
  if (!token) return false;

  const decodedToken = this.jwtHelper.decodeToken(token);
  return decodedToken.role === 'Admin';
}
```

5.5 Exercises

1. Implement JWT authentication for login in ASP.NET Core API.
2. Protect API endpoints using `[Authorize]` and role-based authorization.
3. Configure Angular interceptor to attach JWT token to all API requests.
4. Implement `AuthGuard` in Angular to prevent unauthorized route access.
5. Test login with multiple roles (Admin, User) and verify access.

5.6 Real-World Example

Scenario: Online Product Management System

- Admin can add/edit/delete products
- Regular users can only view products
- Angular interceptors attach JWT tokens to requests
- Guards prevent users from accessing restricted routes

ChatGPT said:

Perfect! ✅ Let's continue with **Section 4 – Chapter 6: CRUD Operations – Full Stack Example**. This chapter focuses on **implementing a full CRUD application using ASP.NET Core APIs and Angular frontend**, covering models, controllers, services, and UI display.

Section 4 – Advanced Topics / Full Stack Integration

Chapter 6: CRUD Operations – Full Stack Example

6.1 Creating Models in ASP.NET Core

Step 1 – Create Product Model

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}
```

Step 2 – Configure DbContext

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
{}
    public DbSet<Product> Products { get; set; }
}
```

Step 3 – Add Database Connection

appsettings.json

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=FullStackDB;Trusted_Connection=True;"
  }
}
```

Program.cs

```
builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection
")));
```

6.2 Implementing API Controllers with CRUD Endpoints

Step 1 – Generate Controller

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Step 2 – API Controller Example

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    private readonly AppDbContext _context;
    public ProductsController(AppDbContext context)
    {
        _context = context;
    }

    [HttpGet]
    public IActionResult GetProducts() => Ok(_context.Products.ToList());

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        var product = _context.Products.Find(id);
        return product == null ? NotFound() : Ok(product);
    }

    [HttpPost]
    public IActionResult AddProduct(Product product)
    {
        _context.Products.Add(product);
        _context.SaveChanges();
        return CreatedAtAction(nameof(GetProduct), new { id = product.Id },
```

```
product);
    }

    [HttpPut("{id}")]
    public IActionResult UpdateProduct(int id, Product product)
    {
        var existing = _context.Products.Find(id);
        if (existing == null) return NotFound();
        existing.Name = product.Name;
        existing.Price = product.Price;
        existing.Description = product.Description;
        _context.SaveChanges();
        return NoContent();
    }

    [HttpDelete("{id}")]
    public IActionResult DeleteProduct(int id)
    {
        var product = _context.Products.Find(id);
        if (product == null) return NotFound();
        _context.Products.Remove(product);
        _context.SaveChanges();
        return NoContent();
    }
}
```

6.3 Connecting Angular Services to API

Step 1 – Create Product Service

```
ng generate service services/product
```

Step 2 – Service Methods

```
@Injectable({ providedIn: 'root' })
export class ProductService {
  private apiUrl = 'https://localhost:5001/api/products';

  constructor(private http: HttpClient) {}

  getProducts() { return this.http.get<Product[]>(this.apiUrl); }
```

```
  getProduct(id: number) { return
this.http.get<Product>(`${this.apiUrl}/${id}`); }
  addProduct(product: Product) { return this.http.post(this.apiUrl, product); }
  updateProduct(id: number, product: Product) { return
this.http.put(`${this.apiUrl}/${id}`, product); }
  deleteProduct(id: number) { return this.http.delete(`${this.apiUrl}/${id}`); }
}
```

6.4 Displaying Data in Angular Components

Step 1 – Product List Component

```
export class ProductListComponent implements OnInit {
  products: Product[] = [];

  constructor(private productService: ProductService) {}

  ngOnInit() {
    this.loadProducts();
  }

  loadProducts() {
    this.productService.getProducts().subscribe(data => this.products = data);
  }

  deleteProduct(id: number) {
    this.productService.deleteProduct(id).subscribe(() => this.loadProducts());
  }
}
```

Step 2 – Product List Template

```
<table>
  <tr>
    <th>Name</th>
    <th>Price</th>
    <th>Description</th>
    <th>Actions</th>
  </tr>
  <tr *ngFor="let product of products">
    <td>{{product.name}}</td>
```

```
    <td>{{product.price}}</td>
    <td>{{product.description}}</td>
    <td>
      <button (click)="editProduct(product)">Edit</button>
      <button (click)="deleteProduct(product.id)">Delete</button>
    </td>
  </tr>
</table>
```

Step 3 – Add/Edit Product Component

- Create a form using **ReactiveFormsModule**
- Bind form fields to service methods for adding/updating products

6.5 Exercises

1. Create a complete CRUD API for a `Category` model.
2. Implement Angular services to consume the `Category` API.
3. Display list of categories with Add/Edit/Delete functionality.
4. Integrate form validation for product creation/editing.
5. Combine product and category data using **nested API calls** in Angular.

6.6 Real-World Example

- **Scenario:** Product Management System
  - Backend API: ASP.NET Core CRUD endpoints for `Product`
  - Angular frontend: ProductListComponent, ProductFormComponent
  - Features: List, Add, Edit, Delete, Search

Chapter 7: File Upload and Download in Full Stack Applications

7.1 Uploading Files from Angular to ASP.NET Core

Step 1 – Create File Upload API in ASP.NET Core

```
[Route("api/[controller]")]
[ApiController]
public class FileController : ControllerBase
{
    private readonly IWebHostEnvironment _env;

    public FileController(IWebHostEnvironment env)
    {
        _env = env;
    }

    [HttpPost("upload")]
    public async Task<IActionResult> Upload(IFormFile file)
    {
        if (file == null || file.Length == 0)
            return BadRequest("No file selected");

        var path = Path.Combine(_env.ContentRootPath, "Uploads", file.FileName);

        using (var stream = new FileStream(path, FileMode.Create))
        {
            await file.CopyToAsync(stream);
        }

        return Ok(new { FileName = file.FileName });
    }
}
```

Step 2 – Create Upload Folder

- Create folder `Uploads` in project root
- Ensure **write permissions**

7.2 Angular File Upload Service

Step 1 – Create Service

```
ng generate service services/file
```

Step 2 – Service Method

```
uploadFile(file: File): Observable<any> {
  const formData = new FormData();
  formData.append('file', file, file.name);
  return this.http.post('https://localhost:5001/api/file/upload', formData);
}
```

Step 3 – Component for File Upload

```
export class FileUploadComponent {
  selectedFile: File | null = null;

  constructor(private fileService: FileService) {}

  onFileSelected(event: any) {
    this.selectedFile = event.target.files[0];
  }

  upload() {
    if (this.selectedFile) {
      this.fileService.uploadFile(this.selectedFile).subscribe(response => {
        console.log('File uploaded:', response);
      });
    }
  }
}
```

Step 4 – Template HTML

```
<input type="file" (change)="onFileSelected($event)">
<button (click)="upload()">Upload</button>
```

7.3 Downloading Files from Backend

Step 1 – ASP.NET Core API for Download

```
[HttpGet("download/{fileName}")]
public IActionResult Download(string fileName)
{
    var path = Path.Combine(_env.ContentRootPath, "Uploads", fileName);
    if (!System.IO.File.Exists(path)) return NotFound();

    var bytes = System.IO.File.ReadAllBytes(path);
    return File(bytes, "application/octet-stream", fileName);
}
```

Step 2 – Angular Download Method

```
downloadFile(fileName: string) {
  return this.http.get(`https://localhost:5001/api/file/download/${fileName}`,
{ responseType: 'blob' });
}

saveFile(fileName: string, blob: Blob) {
  const link = document.createElement('a');
  link.href = window.URL.createObjectURL(blob);
  link.download = fileName;
  link.click();
}
```

Step 3 – Component Method

```
download(fileName: string) {
  this.fileService.downloadFile(fileName).subscribe(blob =>
this.fileService.saveFile(fileName, blob));
}
```

7.4 Handling File Validation and Size Limits

Backend Validation Example:

```
if (file.Length > 5 * 1024 * 1024)
    return BadRequest("File size cannot exceed 5 MB");

var allowedExtensions = new[] { ".jpg", ".png", ".pdf" };
var extension = Path.GetExtension(file.FileName).ToLower();
if (!allowedExtensions.Contains(extension))
```

```
    return BadRequest("Invalid file type");
```

Frontend Validation Example:

```
onFileSelected(event: any) {
  const file = event.target.files[0];
  if (file.size > 5 * 1024 * 1024) {
    alert("File size exceeds 5 MB");
    return;
  }
  const allowedTypes = ["image/jpeg", "image/png", "application/pdf"];
  if (!allowedTypes.includes(file.type)) {
    alert("Invalid file type");
    return;
  }
  this.selectedFile = file;
}
```

7.5 Exercises

1. Implement file upload for user profile images.
2. Validate file type and size in Angular before uploading.
3. Implement download functionality for uploaded files.
4. Extend backend to save file metadata (name, size, type) in database.
5. Display uploaded files in Angular list with download links.

7.6 Real-World Example

Scenario: Document Management System

- Users can upload PDF reports
- File metadata is stored in SQL Server
- Files are downloadable by authorized users
- Angular displays list of uploaded documents

Chapter 8: Advanced Data Handling

8.1 Pagination in ASP.NET Core

Definition:
Pagination splits large datasets into smaller, manageable pages to improve performance and user experience.

Step 1 – API Endpoint with Pagination

```
[HttpGet]
public IActionResult GetProducts(int pageNumber = 1, int pageSize = 10)
{
    var products = _context.Products
        .Skip((pageNumber - 1) * pageSize)
        .Take(pageSize)
        .ToList();

    return Ok(products);
}
```

Step 2 – Frontend Integration

```
getProducts(pageNumber: number, pageSize: number) {
  return
this.http.get<Product[]>(`${this.apiUrl}?pageNumber=${pageNumber}&pageSize=${page
Size}`);
}
```

Exercise:

1. Implement pagination for `Product` list with 10 items per page.
2. Add navigation buttons for previous/next pages in Angular template.

8.2 Sorting Data

Backend Sorting Example:

```
[HttpGet("sort")]
public IActionResult GetProductsSorted(string sortBy = "Name")
{
    var products = sortBy.ToLower() switch
    {
        "price" => _context.Products.OrderBy(p => p.Price).ToList(),
        _ => _context.Products.OrderBy(p => p.Name).ToList()
    };

    return Ok(products);
}
```

Angular Service for Sorting:

```
getSortedProducts(sortBy: string) {
  return this.http.get<Product[]>(`${this.apiUrl}/sort?sortBy=${sortBy}`);
}
```

Exercise:

1. Implement sorting by `Name` and `Price`.
2. Add sort dropdown in Angular component.

8.3 Filtering Data

Backend Filtering Example:

```
[HttpGet("filter")]
public IActionResult FilterProducts(string name = "")
{
    var products = _context.Products
        .Where(p => p.Name.Contains(name))
        .ToList();

    return Ok(products);
}
```

Angular Service for Filtering:

```
filterProducts(name: string) {
  return this.http.get<Product[]>(`${this.apiUrl}/filter?name=${name}`);
```

```
}
```

Exercise:

1. Implement a search box in Angular to filter products by name.
2. Display filtered results dynamically without reloading the page.

8.4 Combining Pagination, Sorting, and Filtering

Backend Example:

```
[HttpGet("advanced")]
public IActionResult GetProductsAdvanced(int pageNumber = 1, int pageSize = 10,
string sortBy = "Name", string name = "")
{
    var query = _context.Products.AsQueryable();

    if (!string.IsNullOrEmpty(name))
        query = query.Where(p => p.Name.Contains(name));

    query = sortBy.ToLower() switch
    {
        "price" => query.OrderBy(p => p.Price),
        _ => query.OrderBy(p => p.Name)
    };

    var products = query.Skip((pageNumber - 1) * pageSize)
                        .Take(pageSize)
                        .ToList();

    return Ok(products);
}
```

Angular Integration:

- Use dropdowns for sort options
- Input box for search/filter
- Pagination buttons to navigate pages

Exercise:

1. Implement combined pagination, sorting, and filtering for products.
2. Ensure UI updates dynamically based on user interaction.

8.5 Real-World Example

Scenario: E-commerce Product Listing

- Users can search products by name
- Sort products by price or name
- Paginate results to improve performance
- Angular dynamically fetches and displays data from ASP.NET Core API

8.6 Exercises Summary

1. Implement server-side pagination for product API.
2. Add sorting functionality in both backend and frontend.
3. Implement search/filter using API query parameters.
4. Combine pagination, sorting, and filtering in a single component.
5. Test performance with large datasets (1000+ records).

Chapter 9: Full Stack Deployment and Best Practices

9.1 Preparing Angular Application for Production

Step 1 – Build Angular for Production

```
ng build --prod
```

- Creates a `dist/FullStackFrontend` folder
- Minifies and optimizes HTML, CSS, and JS files for production

Step 2 – Configure Base HREF

- Ensure correct `base href` in `index.html`

```
<base href="/">
```

Step 3 – Serve Angular App via ASP.NET Core

- Copy `dist` folder to `wwwroot` in ASP.NET Core project
- Add middleware in `Program.cs`

```
app.UseDefaultFiles();
app.UseStaticFiles();
```

9.2 Deploying ASP.NET Core API

Step 1 – Configure Production Database

- Update `appsettings.json` with production connection string

```
"ConnectionStrings": {
    "DefaultConnection": "Server=ProdServer;Database=FullStackDB;User
Id=sa;Password=YourPassword;"
}
```

Step 2 – Publish API

```
dotnet publish -c Release -o ./publish
```

- Creates compiled binaries for deployment

Step 3 – Deploy to IIS

1. Install **.NET Hosting Bundle** on server
2. Create a new website in IIS
3. Point physical path to `publish` folder
4. Configure **Application Pool** with `No Managed Code`

Step 4 – Enable CORS for Frontend

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAngular", policy =>
        policy.WithOrigins("https://yourfrontenddomain.com")
              .AllowAnyMethod()
```

```
            .AllowAnyHeader());
});
```

9.3 Environment Configurations

Angular Environment Files

- `environment.ts` – Development
- `environment.prod.ts` – Production

Example:

```
export const environment = {
  production: true,
  apiUrl: 'https://api.yourdomain.com'
};
```

ASP.NET Core Environment Variables

- `ASPNETCORE_ENVIRONMENT=Production`
- Use `appsettings.Production.json` for production configs

9.4 Security Best Practices

1. **Use HTTPS** – Enforce SSL on backend and frontend
2. **JWT Expiration & Refresh Tokens** – Prevent token misuse
3. **Data Validation** – Validate both client-side and server-side
4. **Error Handling** – Avoid exposing internal exceptions
5. **CORS Restrictions** – Only allow trusted origins
6. **Logging** – Track errors and critical actions using Serilog/NLog

9.5 Performance Optimization

1. **Angular**
    a. Lazy load modules
    b. Minify CSS/JS
    c. Enable production mode (`enableProdMode()`)
2. **ASP.NET Core**

a. Use **Response Caching**
b. Minimize database queries
c. Enable **Gzip compression**
d. Use **Asynchronous Controllers** for heavy I/O operations

9.6 Deployment Exercises

1. Build Angular app for production and serve via ASP.NET Core
2. Deploy ASP.NET Core API to IIS or any cloud server
3. Configure production environment settings for API and Angular
4. Test HTTPS, JWT security, and CORS configuration
5. Optimize performance by testing load times and API response

9.7 Real-World Example

Scenario: Online Store Application

- Angular app served via ASP.NET Core `wwwroot`
- API deployed on cloud server with HTTPS
- Users access production environment securely
- Lazy loading ensures fast frontend load
- JWT authentication protects APIs

Chapter 10: Full Stack Testing and Debugging

10.1 Introduction to Testing

Definition:

- **Testing** ensures software works as expected and identifies bugs before production.
- **Types of Testing:**
  - Unit Testing – Test individual components or functions
  - Integration Testing – Test interactions between components or modules
  - End-to-End (E2E) Testing – Test the entire application workflow

10.2 Unit Testing in ASP.NET Core

Step 1 – Install Testing Framework

```
dotnet add package xunit
dotnet add package xunit.runner.visualstudio
dotnet add package Moq
```

Step 2 – Create Test Project

```
dotnet new xunit -n FullStack.Tests
```

Step 3 – Example Unit Test for ProductService

```
public class ProductServiceTests
{
    private readonly Mock<IUnitOfWork> _mockUnitOfWork;
    private readonly ProductService _productService;

    public ProductServiceTests()
    {
        _mockUnitOfWork = new Mock<IUnitOfWork>();
        _productService = new ProductService(_mockUnitOfWork.Object);
    }

    [Fact]
    public void GetProducts_ReturnsAllProducts()
    {
        // Arrange
        _mockUnitOfWork.Setup(u => u.Products.GetAll()).Returns(new List<Product>
        {
            new Product { Id = 1, Name = "Laptop", Price = 1000 },
            new Product { Id = 2, Name = "Mouse", Price = 50 }
        });

        // Act
        var result = _productService.GetProducts();

        // Assert
        Assert.Equal(2, result.Count());
    }
```

```
}
```

Exercise:

1. Write unit tests for `AddProduct` and `DeleteProduct` methods.
2. Test repository layer using `Moq` to simulate database operations.

10.3 Unit Testing in Angular

Step 1 – Jasmine and Karma

- Angular projects come with Jasmine (testing framework) and Karma (test runner)

Step 2 – Example Test for ProductService

```
describe('ProductService', () => {
  let service: ProductService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [ProductService]
    });

    service = TestBed.inject(ProductService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  it('should fetch products', () => {
    const dummyProducts = [
      { id: 1, name: 'Laptop', price: 1000, description: 'Gaming Laptop' },
      { id: 2, name: 'Mouse', price: 50, description: 'Wireless Mouse' }
    ];

    service.getProducts().subscribe(products => {
      expect(products.length).toBe(2);
      expect(products).toEqual(dummyProducts);
    });

    const req = httpMock.expectOne('https://localhost:5001/api/products');
    expect(req.request.method).toBe('GET');
```

```
    req.flush(dummyProducts);
  });
});
```

Exercise:

1. Write unit tests for `addProduct` and `deleteProduct` methods.
2. Test error handling in Angular HTTP calls.

10.4 Integration Testing

Definition:

- Ensures **multiple components work together correctly**
- Examples: Angular components with services, API endpoints with database

ASP.NET Core Example:

```
[Fact]
public async Task GetProducts_ReturnsOkResult()
{
    using var factory = new WebApplicationFactory<Program>();
    var client = factory.CreateClient();

    var response = await client.GetAsync("/api/products");

    response.EnsureSuccessStatusCode();
    var products = JsonConvert.DeserializeObject<List<Product>>(await
response.Content.ReadAsStringAsync());
    Assert.NotNull(products);
}
```

Exercise:

1. Test API endpoints for CRUD operations using integration tests.
2. Ensure frontend components fetch and display API data correctly.

10.5 End-to-End Testing with Cypress

Step 1 – Install Cypress

```
npm install cypress --save-dev
```

Step 2 – Example E2E Test

```
describe('Product Management', () => {
  it('should display product list', () => {
    cy.visit('http://localhost:4200/products');
    cy.get('table tr').should('have.length.greaterThan', 1);
  });

  it('should add a new product', () => {
    cy.get('button#addProduct').click();
    cy.get('input#name').type('Keyboard');
    cy.get('input#price').type('30');
    cy.get('textarea#description').type('Mechanical Keyboard');
    cy.get('button#save').click();
    cy.contains('Keyboard').should('exist');
  });
});
```

Exercise:

1. Create E2E tests for login functionality.
2. Test full CRUD flow in Angular + ASP.NET Core.

10.6 Debugging Techniques

Angular Debugging:

- Use `console.log()` for quick debugging
- Use **Angular DevTools** to inspect component state and events
- Debug with Chrome Developer Tools

ASP.NET Core Debugging:

- Use Visual Studio Debugger to set **breakpoints**
- Inspect variable values and call stack
- Use **Logging** to trace errors (`ILogger`)

10.7 Exercises Summary

1. Write unit tests for both backend and frontend services
2. Perform integration testing for APIs and Angular services
3. Implement E2E testing using Cypress
4. Debug common errors using Visual Studio and Chrome DevTools
5. Verify data flow between Angular components and ASP.NET Core APIs

ChatGPT said:

Perfect! ✅ Let's continue with **Section 4 – Chapter 11: Full Stack Project – Capstone Project**. This chapter focuses on **implementing a complete end-to-end project using Angular and ASP.NET Core**, integrating all concepts learned in previous chapters.

🟩 Section 4 – Advanced Topics / Full Stack Integration

Chapter 11: Full Stack Project – Capstone Project

11.1 Project Overview

Project Name: Online Product Management System

Project Features:

1. User Registration and Login (JWT Authentication)
2. Role-Based Access (Admin/User)
3. Product CRUD Operations (Create, Read, Update, Delete)
4. File Upload and Download (Product Images)
5. Pagination, Sorting, and Filtering of Products
6. Logging and Error Handling

7. Deployment Ready (Production Build + Environment Configuration)

Technologies Used:

- Backend: ASP.NET Core 8, Entity Framework Core, SQL Server
- Frontend: Angular 17, TypeScript, RxJS
- Tools: Visual Studio, VS Code, Postman, Git

11.2 Backend Implementation

Step 1 – Create ASP.NET Core API Project

```
dotnet new webapi -n FullStackAPI
```

Step 2 – Configure Models and DbContext

- Models: `User`, `Product`, `Category`, `FileMetadata`
- DbContext: `AppDbContext` with `DbSet` properties for all models

Step 3 – Implement Authentication and Authorization

- JWT Authentication
- Role-based Authorization for Admin and Users

Step 4 – Implement Product and Category CRUD

- Controllers: `ProductsController`, `CategoriesController`
- Repository and UnitOfWork Patterns

Step 5 – File Upload and Download API

- Controller: `FileController`
- Save files in `Uploads` folder and metadata in database

Step 6 – Logging and Error Handling

- Use `ILogger` for backend logging
- Global exception handling middleware

11.3 Frontend Implementation

Step 1 – Create Angular Project

```
ng new FullStackFrontend --routing --style=scss
```

Step 2 – Create Angular Modules and Components

- Modules: `AuthModule`, `ProductModule`, `SharedModule`
- Components: `LoginComponent`, `RegisterComponent`, `ProductListComponent`, `ProductFormComponent`, `FileUploadComponent`

Step 3 – Services for API Communication

- `AuthService` – Handle login, registration, JWT token storage
- `ProductService` – CRUD operations for products
- `CategoryService` – CRUD operations for categories
- `FileService` – Upload and download files

Step 4 – Implement Routing and Guards

- Use `AuthGuard` to restrict unauthorized access
- Route roles-based access for Admin and Users

Step 5 – UI Implementation

- Display products with pagination, sorting, and filtering
- Forms for adding/editing products with validation
- File upload/download interface

11.4 Integrating Backend and Frontend

1. Configure Angular environment for API URLs (`environment.prod.ts`)
2. Attach JWT token in HTTP headers using **HTTP Interceptor**
3. Test API endpoints using **Postman**
4. Bind Angular components to API data dynamically using `HttpClient`
5. Handle errors gracefully with alert messages or notification components

11.5 Deployment

1. Build Angular app for production (`ng build --prod`)
2. Serve Angular via ASP.NET Core `wwwroot` or separate hosting
3. Deploy backend API to IIS or cloud server
4. Configure CORS, HTTPS, and environment variables
5. Test full-stack application end-to-end

11.6 Exercises

1. Implement a full product management system with users and roles
2. Upload product images and associate with product records
3. Implement search, filter, sort, and pagination in the product list
4. Ensure proper logging and error handling for backend APIs
5. Deploy the application and verify production readiness

11.7 Real-World Example

Scenario:

- An e-commerce store allows admins to manage products and categories
- Users can browse products, view details, and download product documents
- Angular frontend consumes APIs securely using JWT authentication
- Full-stack project integrates all advanced concepts from Sections 1–4

Chapter 12: Full Stack Best Practices and Optimization

12.1 Coding Standards and Clean Code

Definition:

- Writing **readable, maintainable, and consistent code** for long-term projects.

Best Practices:

1. **Naming Conventions:**
    a. Classes: PascalCase (`ProductService`)
    b. Methods: PascalCase (`GetProducts()`)
    c. Variables: camelCase (`productList`)
2. **File Organization:**
    a. Keep Angular components, services, and modules organized by feature
    b. Use ASP.NET Core folders like `Controllers`, `Models`, `Services`, `Data`
3. **Single Responsibility Principle (SRP):**
    a. Each class or component should have **one responsibility**
4. **Code Reusability:**
    a. Use Angular shared modules for common components
    b. Use backend service classes for repeated API logic

Exercise:

- Refactor an existing Angular component to follow SRP and modularization

12.2 Security Best Practices

1. Authentication & Authorization

- Use JWT tokens for API authentication
- Implement role-based access control for sensitive operations

2. HTTPS & CORS

- Always serve apps over HTTPS
- Restrict CORS to trusted domains

3. Input Validation & Sanitization

- Validate user inputs both on **frontend and backend**
- Prevent SQL Injection and XSS attacks

4. Error Handling

- Do not expose internal errors to end-users
- Log errors on backend using `ILogger` or third-party tools (e.g., Serilog)

Exercise:

- Secure all APIs in a demo project and implement error logging

12.3 Performance Optimization

Frontend (Angular):

1. **Lazy Loading Modules:** Reduce initial load time
2. **OnPush Change Detection:** Improve component rendering performance
3. **AOT Compilation:** Angular Ahead-of-Time compilation for faster startup
4. **Minification & Bundling:** Reduce file size for production

Backend (ASP.NET Core):

1. **Asynchronous Programming:** Use async/await for API calls
2. **Caching:** Use response caching for frequent API calls
3. **Database Optimization:**
    a. Use proper indexing
    b. Avoid unnecessary queries
4. **Gzip Compression:** Reduce response size

Exercise:

- Implement lazy loading for product modules
- Enable response caching for product listing API

12.4 Maintainability and Scalability

1. Modular Architecture:

- Separate features into modules in Angular
- Use layered architecture in backend: Controllers → Services → Repositories

2. Version Control:

- Use Git with proper branching strategy (main, dev, feature branches)

3. Documentation:

- Document API endpoints using **Swagger**
- Comment complex code logic for future reference

4. Automated Testing:

- Use unit tests, integration tests, and E2E tests
- Run tests as part of CI/CD pipeline

Exercise:

- Set up Swagger documentation for APIs
- Implement a feature branch workflow in Git for a demo project

12.5 Real-World Example

Scenario:

- Large e-commerce project
- Multiple developers working on Angular frontend and ASP.NET Core backend
- Implemented coding standards, modular architecture, caching, security, and logging
- CI/CD pipeline ensures smooth deployment and testing