**C#-ASP.NET**

It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.

C# has roots from the C family, and the language is close to other popular languages like [C++](https://www.w3schools.com/cpp/default.asp) and [Java](https://www.w3schools.com/java/default.asp).

**Case-sensitive** (like Java, C++, etc.)

The first version was released in year 2002. The latest version, **C# 13**, was released in November 2024.

C# is used for:

* Mobile applications
* Desktop applications
* Web applications
* Web services
* Web sites
* Games
* VR
* Database applications
* And much, much more!

Program.cs

using System;

namespace HelloWorld

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}

* using System; – includes the System namespace which contains basic classes like Console.
* namespace – groups related classes.
* class Program – defines a class (like a blueprint).
* Main() – the entry point of a C# application

There is also a Write() method, which is similar to WriteLine().

The only difference is that it does not insert a new line at the end of the output:

Single-line comments start with two forward slashes (//).  
Multi-line comments start with /\* and ends with \*/.

Any text between /\* and \*/ will be ignored by C#.

# C# Variables

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

* int - stores integers (whole numbers), without decimals, such as 123 or -123
* double - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* string - stores text, such as "Hello World". String values are surrounded by double quotes
* bool - stores values with two states: true or false

Stored directly in memory. Examples:

int age = 30;

double salary = 50000.75;

char grade = 'A';

bool isValid = true;

# C# Constants

# If you don't want others (or yourself) to overwrite existing values, you can add the const keyword in front of the variable type.This will declare the variable as "constant", which means unchangeable and read-only:

const int myNum = 15;

myNum = 20; // error

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

const int myNum = 15;

myNum = 20;

Console.WriteLine(myNum);

}

}

}

// prog.cs(10,7): error CS0131: The left-hand side of an assignment must be a variable, a property or an indexer

Compilation failed: 1 error(s), 0 warnings7): error CS0131: The left-hand side of an assignment must be a variable, a property or an indexer

# C# Display Variables

The WriteLine() method is often used to display variable values to the console window.

To combine both text and a variable, use the + character:

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

string name = "John";

Console.WriteLine("Hello " + name);

}}} // Hello John

Lat/ion failed: 1 error(s), 0 warnings

# C# Multiple Variables

To declare more than one variable of the **same type**, use a comma-separated list:

int x = 5, y = 6, z = 50;

Console.WriteLine(x + y + z) // 61

# C# Identifiers

All C# **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

It is recommended to use descriptive names in order to create understandable and maintainable code:

The general rules for naming variables are:

* Names can contain letters, digits and the underscore character (\_)
* Names must begin with a letter or underscore
* Names should start with a lowercase letter, and cannot contain whitespace
* Names are case-sensitive ("myVar" and "myvar" are different variables)
* Reserved words (like C# keywords, such as int or double) cannot be used as names

int minutesPerHour = 60; ( Good)

int m = 60;(OK, but not so easy to understand what m actually is)

# C# Data Types

A data type specifies the size and type of variable values.

It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

| **Data Type** | **Size (in bytes)** | **Description** |
| --- | --- | --- |
| int | 4 bytes | Stores whole numbers from **−2,147,483,648 to 2,147,483,647**. Commonly used for counting or loops. |
| long | 8 bytes | Stores large whole numbers from **−9 quintillion to +9 quintillion** (specifically −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). |
| float | 4 bytes | Stores **fractional numbers** (decimals) with **6–7 digits of precision**. Use f suffix: float pi = 3.14f; |
| double | 8 bytes | Stores fractional numbers with **about 15–16 digits of precision**. Use by default for decimals. |
| bool | 1 byte | Stores true or false. Used in logical conditions. |
| char | 2 bytes (Unicode) | Stores a single character (e.g., 'A', '9', '#'). Use **single quotes**. |
| string | 2 bytes per character | Stores a **sequence of characters** like words or sentences (e.g., "Hello"). Use **double quotes**. Internally, each character is a Unicode char. |

**C# Type Casting**  
Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

* **Implicit Casting** (automatically) - converting a smaller type to a larger type size  
  char -> int -> long -> float -> double
* **Explicit Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char

Implicit casting is done automatically when passing a smaller size type to a larger size type:

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

Console.WriteLine(myInt); // Outputs 9

Console.WriteLine(myDouble)

Explicit casting must be done manually by placing the type in parentheses in front of the value:

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

Console.WriteLine(myDouble); // Outputs 9.78

Console.WriteLine(myInt);

## Type Conversion Methods

## It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

int myInt = 10;

double myDouble = 5.25;

bool myBool = true;

Console.WriteLine(Convert.ToString(myInt)); // convert int to string

Console.WriteLine(Convert.ToDouble(myInt)); // convert int to double

Console.WriteLine(Convert.ToInt32(myDouble)); // convert double to int

Console.WriteLine(Convert.ToString(myBool)); // convert bool to string

## Get User Input

## already learned that Console.WriteLine() is used to output (print) values. Now we will use Console.ReadLine() to get user input. Used to **read user input** from the console as a **string**.

The Console.ReadLine() method returns a string. Therefore, you cannot get information from another data type, such as int. The following program will cause an error:

Example

Console.WriteLine("Enter your age:");

int age = Console.ReadLine();

Console.WriteLine("Your age is: " + age);

The error message will be something like this:

Cannot implicitly convert type 'string' to 'int'

# C# Operators

## Arithmetic Operators Arithmetic operators are used to perform common mathematical operations:

| **Operator** | **Name** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| + | Addition | Adds two operands | x + y | If x=5, y=3 → 8 |
| - | Subtraction | Subtracts the right operand from the left | x - y | 2 |
| \* | Multiplication | Multiplies both operands | x \* y | 15 |
| / | Division | Divides the left operand by the right | x / y | 1 (integer division if both are int) |
| % | Modulus | Returns the remainder of division | x % y | 2 |
| ++ | Increment | Increases variable by 1 (can be prefix or postfix) | x++ or ++x | x = 6 |
| -- | Decrement | Decreases variable by 1 (can be prefix or postfix) | x-- or --x | x = 4 |

# C# Assignment Operators

| **Operator** | **Example** | **Same As** | **Description** |
| --- | --- | --- | --- |
| = | x = 5 | x = 5 | Assigns the value 5 to variable x. |
| += | x += 3 | x = x + 3 | Adds 3 to x and stores the result in x. |
| -= | x -= 3 | x = x - 3 | Subtracts 3 from x and stores the result in x. |
| \*= | x \*= 3 | x = x \* 3 | Multiplies x by 3 and stores the result in x. |
| /= | x /= 3 | x = x / 3 | Divides x by 3 and stores the result in x. |
| %= | x %= 3 | x = x % 3 | Assigns the remainder of x divided by 3 to x. |
| &= | x &= 3 | x = x & 3 | Performs a **bitwise AND** on x and 3, and assigns the result to x. |
| ` | =` | `x | = 3` |
| ^= | x ^= 3 | x = x ^ 3 | Performs a **bitwise XOR** on x and 3, and assigns the result to x. |
| >>= | x >>= 3 | x = x >> 3 | Shifts the bits of x **right by 3 places** and assigns the result to x. |
| <<= | x <<= 3 | x = x << 3 | Shifts the bits of x **left by 3 places** and assigns the result to x. |

## Comparison Operators

| **Operator** | **Name** | **Example** | **Description** |
| --- | --- | --- | --- |
| == | Equal to | x == y | Returns true if x is **equal** to y. |
| != | Not equal to | x != y | Returns true if x is **not equal** to y. |
| > | Greater than | x > y | Returns true if x is **greater** than y. |
| < | Less than | x < y | Returns true if x is **less** than y. |
| >= | Greater than or equal to | x >= y | Returns true if x is **greater than or equal to** y. |
| <= | Less than or equal to | x <= y | Returns true if x is **less than or equal to** y. |

## Logical Operators

| **Operator** | **Name** | **Example** | **Description** |
| --- | --- | --- | --- |
| && | Logical AND | x < 5 && x < 10 | Returns true **only if both** conditions are true. |
| || | Logical OR | x < 5 || x < 4 | Logical OR |
| ! | Logical NOT | !(x < 5 && x < 10) | **Inverts** the result: true becomes false, and vice versa. |

## C# If ... Else C# Conditions and If Statements

C# supports the usual logical conditions from mathematics:

* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b
* Equal to a == b
* Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

C# has the following conditional statements:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed

## The if Statement

Use the if statement to specify a block of C# code to be executed if a condition is True.

If(condition)

{  
// block of code to be executed if the condition is True

}

Eg: if (20 > 18)

{

Console.WriteLine("20 is greater than 18");

}//output is 20 is greater than 18

## The else Statement

Use the else statement to specify a block of code to be executed if the condition is False.  
if (condition)

{

// block of code to be executed if the condition is True

}

else

{

// block of code to be executed if the condition is False

# }}

# C# Break and Continue

The break statement can also be used to jump out of a **loop**.

This example jumps out of the loop when i is equal to 4:

for (int i = 1; i <= 10; i++)

{

if (i == 5)

break;

Console.WriteLine(i);

}  
  
Output

1

2

3

4

⛔ The loop stops when i == 5 and does **not** print 5–10.

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

for (int i = 1; i <= 5; i++)

{

if (i == 4)

continue;

Console.WriteLine(i);

}  
Output:

1

2

3

5

The value 4 is skipped.

**C# Arrays**  
Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:  
string[] cars;

We have now declared a variable that holds an array of strings.

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:  
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

To create an array of integers, you could write:  
int[] myNum = {10, 20, 30, 40};

You access an array element by referring to the index number.

This statement accesses the value of the first element in **cars**:  
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars[0]);

// Outputs Volvo

## Loop Through an Array

You can loop through the array elements with the for loop, and use the Length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.Length; i++)

{

Console.WriteLine(cars[i]);

}

## The foreach Loop There is also a foreach loop, which is used exclusively to loop through elements in an **array**: foreach (type variableName in arrayName)

## {

## // code block to be executed

## }

### C# Methods

### A **method** is a **block of code** that performs a specific task. It helps you **organize code**, **reuse logic**, and make your program **modular**. ****1. Defining a Method****

returnType MethodName(parameter1Type parameter1, ...)

{

// method body

return value; // if returnType is not void

}

#### 🧪 Example:

int Add(int a, int b)

{

return a + b;

}

### 🔹 ****2. Calling a Method****

Use the method name and pass arguments:

int sum = Add(5, 3);

Console.WriteLine("Sum is: " + sum);

### 🔹 ****3. Types of Methods****

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Void Method** | Doesn’t return anything | void SayHello() { Console.WriteLine("Hi"); } |
| **Return Method** | Returns a value (int, string, etc.) | int Square(int x) { return x \* x; } |
| **Static Method** | Belongs to the class, not object | static int Multiply(int a, int b) |
| **Instance Method** | Needs object to be called | car.Start(); where Start() is an instance method |
| **Parameterized** | Accepts input values | Greet(string name) |
| **Parameterless** | Doesn’t accept input values | Display() |

### 🔹 ****4. Method Overloading****

Same method name, different parameters.

void Greet() {

Console.WriteLine("Hello");

}

void Greet(string name) {

Console.WriteLine($"Hello, {name}");

}

### 🔹 ****5. Static vs Instance Methods****

class MathUtils

{

public static int Add(int a, int b) => a + b; // Static method

public int Multiply(int a, int b) => a \* b; // Instance method

}

// Call static method directly using class name

int result1 = MathUtils.Add(2, 3);

// Call instance method via object

MathUtils mu = new MathUtils();

int result2 = mu.Multiply(4, 5);

### 🔹 ****6. Method with Default Parameters****

void PrintInfo(string name = "Guest")

{

Console.WriteLine("Welcome " + name);

}

Calling PrintInfo(); → outputs: Welcome Guest

### 🔹 ****7. Method with**** ref ****and**** out

#### ref – passes variable ****by reference****, value must be initialized:

void AddFive(ref int x)

{

x += 5;}

#### out – passes variable ****by reference****, value will be assigned inside:

void GetValues(out int a, out int b)

{

a = 10;

b = 20;

}

### 🔹 ****8. Expression-bodied Method****

Short form using =>:

int Square(int x) => x \* x;

### 🧪 ****Full Example****

class Program

{

static void SayHello(string name)

{

Console.WriteLine($"Hello, {name}!");

}

static int Multiply(int a, int b)

{

return a \* b;

}

static void Main()

{

SayHello("Anagha");

int result = Multiply(5, 3);

Console.WriteLine("5 \* 3 = " + result);

}

}

C# OOP  
OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

## Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create an Object

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new:

# class Car

# {

# string color = "red";

# static void Main(string[] args)

# {

# Car myObj = new Car();

# Console.WriteLine(myObj.color);

# }

# }

# C# Constructors

A **constructor** is a **special method** that is **automatically called** when an object of a class is **created**.  
Its purpose is to **initialize** the object.

## ✅ Key Features of Constructors

* Name is **same as the class name**
* **No return type** (not even void)
* Called **automatically** when an object is created
* Can be **overloaded** like regular methods

## 🧱 Basic Syntax

csharp

CopyEdit

class ClassName

{

public ClassName()

{

// constructor code

}

}

## 🔸 1. ****Default Constructor****

A constructor **without parameters**. If you don’t write one, the compiler creates it automatically.

csharp

CopyEdit

class Car

{

public Car()

{

Console.WriteLine("Default constructor called");

}

}

**Usage:**

csharp

CopyEdit

Car c = new Car(); // Output: Default constructor called

## 🔸 2. ****Parameterized Constructor****

A constructor that takes arguments. Useful for **custom initialization**.

csharp

CopyEdit

class Person

{

public string Name;

public Person(string name)

{

Name = name;

}

}

**Usage:**

csharp

CopyEdit

Person p = new Person("Anagha");

Console.WriteLine(p.Name); // Output: Anagha

## 🔸 3. ****Constructor Overloading****

You can have **multiple constructors** with different parameter sets.

class Student

{

public string Name;

public int Age;

public Student() // Default

{

Name = "Unknown";

Age = 0;

}

public Student(string name) // One parameter

{

Name = name;

Age = 18;

}

public Student(string name, int age) // Two parameters

{

Name = name;

Age = age;

}

}

**Usage:**

Student s1 = new Student();

Student s2 = new Student("Anagha");

Student s3 = new Student("Anagha", 22);

## 🔸 4. ****Static Constructor****

* Used to initialize **static data**
* **Runs only once** for the whole class
* **No parameters allowed**

class Demo

{

static Demo()

{

Console.WriteLine("Static constructor called");

}

public Demo()

{

Console.WriteLine("Instance constructor called");

}

}

**Usage:**

Demo d = new Demo();

// Output:

// Static constructor called

// Instance constructor called

## 🔸 5. ****Private Constructor****

Used to **restrict instantiation** from outside. Often used in **Singleton Pattern**.

class Secret

{

private Secret()

{

Console.WriteLine("Private constructor");

}

public static void Show()

{

Console.WriteLine("Static method in a private constructor class");

}

}

## 🧾 Summary Table

| **Constructor Type** | **Description** |
| --- | --- |
| Default | No parameters |
| Parameterized | Accepts arguments for custom init |
| Overloaded | Multiple versions with different params |
| Static | Initializes static members, runs once |
| Private | Used to restrict object creation |

## 🧪 Quick Example Recap

class Book

{

public string Title;

// Parameterized constructor

public Book(string title)

{

Title = title;

}

}

Book b1 = new Book("C# Basics");

Console.WriteLine(b1.Title); // Output: C# Basics

# C# Access Modifiers

**Access modifiers** in C# define the **visibility** or **scope** of a class, method, property, field, or constructor — i.e., **who can access it**.

## ✅ C# Access Modifiers Overview

| **Modifier** | **Access Level** | **Accessible From** |
| --- | --- | --- |
| public | No restriction | Anywhere (inside or outside the class/project) |
| private | Most restricted | Inside the **same class only** |
| protected | In the same class **or** derived class | Only within class and its subclasses |
| internal | Accessible within the **same assembly/project** | Other classes in the same project |
| protected internal | protected + internal | Same assembly **OR** any subclass (even outside project) |
| private protected | private + protected | Same class or derived class **within the same project only** |

## 🔸 1. public – Accessible Everywhere

public class Car

{

public string brand = "Toyota";

}

**Usage:**

Car c = new Car();

Console.WriteLine(c.brand); // ✅ Works anywhere

## 🔸 2. private – Only Within the Same Class

class BankAccount

{

private double balance = 1000;

public void ShowBalance()

{

Console.WriteLine(balance); // ✅ Allowed

}

}

**Usage:**

BankAccount acc = new BankAccount();

// Console.WriteLine(acc.balance); ❌ Not allowed

acc.ShowBalance(); // ✅ Allowed

## 🔸 3. protected – Only in Same Class or Subclass

class Animal

{

protected void Speak()

{

Console.WriteLine("Animal speaks");

}

}

class Dog : Animal

{

public void Bark()

{

Speak(); // ✅ Allowed in subclass

}

}

## 🔸 4. internal – Only Inside the Same Assembly (Project)

internal class Helper

{

internal void ShowHelp()

{

Console.WriteLine("Help method");

}

}

✅ Accessible in the same project  
❌ Not accessible from a **different DLL or assembly**

## 🔸 5. protected internal – Protected + Internal

class A

{

protected internal void Display()

{

Console.WriteLine("Protected Internal");

}

}

class B : A

{

void Show()

{

Display(); // ✅ Allowed (in subclass)

}

}

✅ Accessible in same project  
✅ Also accessible in a **derived class** from another project

## 🔸 6. private protected – Private + Protected

* Only accessible in the same class **or** a subclass **in the same project**

class Parent

{

private protected void Message()

{

Console.WriteLine("Private Protected");

}

}

class Child : Parent

{

void Show()

{

Message(); // ✅ Allowed (same project)

}

}

❌ Not allowed in subclasses from **other projects**

## 🧾 Summary Table (Quick View)

| **Modifier** | **Class** | **Same Project** | **Derived Class** | **Outside Project** |
| --- | --- | --- | --- | --- |
| public | ✅ | ✅ | ✅ | ✅ |
| private | ✅ | ❌ | ❌ | ❌ |
| protected | ✅ | ❌ | ✅ | ✅ (only if subclass) |
| internal | ✅ | ✅ | ✅ | ❌ |
| protected internal | ✅ | ✅ | ✅ | ✅ (if subclass) |
| private protected | ✅ | ✅ | ✅ | ❌ |

## 🧪 Example Code (All Together)

public class Demo

{

private int a = 1;

protected int b = 2;

internal int c = 3;

protected internal int d = 4;

private protected int e = 5;

public int f = 6;

public void Show()

{

Console.WriteLine($"a={a}, b={b}, c={c}, d={d}, e={e}, f={f}");

}

}

# C# Inheritance

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

* **Derived Class** (child) - the class that inherits from another class
* **Base Class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

# C# Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/cs/cs_inheritance.php) lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

**C# Abstraction**Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/cs/cs_interface.php) (which you will learn more about in the next chapter).

The abstract keyword is used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods

**C# Interface**  
Another way to achieve [abstraction](https://www.w3schools.com/cs/cs_abstract.php) in C#, is with interfaces.

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):  
An **interface** is like a **contract** — it defines **what a class must do**, but **not how**.

✅ An interface contains **method signatures** (no implementation).  
✅ A class that implements an interface **must provide its own implementation**.

## 🧠 Why Use Interfaces?

* To achieve **abstraction**
* To support **multiple inheritance** (C# doesn't support multiple base classes)
* To create **loosely coupled code**
* To support **polymorphism**

## C# Enums

An enum is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma:

enum Level

{

Low,

Medium,

High

}

## Enum inside a Class

You can also have an enum inside a class

class Program{

enum Level {

Low,

Medium,

High}

static void Main(string[] args)**{**

Level myVar = Level.Medium;

# Console.WriteLine(myVar);

# }

# }

**C# Exceptions**  
When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an **exception** (throw an error).

In C#, **exception handling** is a mechanism to detect, catch, and manage errors that occur during **program execution** without crashing the program.

**C# try and catch**  
The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

## The try and catch keywords come in pairs: try

## {

## // Block of code to try

## }

## catch (Exception e)

## {

// Block of code to handle errors

## Finally The finally block ****always executes****, regardless of whether an exception occurred or not. try

## {

## int[] myNumbers = {1, 2, 3};

## Console.WriteLine(myNumbers[10]);

## }

## catch (Exception e)

## {

## Console.WriteLine("Something went wrong.");

## }

## finally

## {

## Console.WriteLine("The 'try catch' is finished.");

## }

The output will be:

//Something went wrong.  
//The 'try catch' is finished.

## throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception class**. There are many exception classes available in C#: ArithmeticException, FileNotFoundException, IndexOutOfRangeException, TimeOutException, etc:

static void checkAge(int age)

{

if (age < 18)

{

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

}

else

{

Console.WriteLine("Access granted - You are old enough!");

}

}

static void Main(string[] args)

{

checkAge(15);

}

The error message displayed in the program will be:

System.ArithmeticException: 'Access denied - You must be at least 18 years old.'

**INVERSION OF CONTROL (IoC)**

Inversion of Control is a design principle used in object-oriented programming to achieve loose coupling between classes and their dependencies.

Normally, a class creates its own dependencies using new.

Traditional Control Flow(No IoC):A high-level module like controller is responsible for creating and managing other service.

With IoC : The responsibility for creating and providing dependencies given to external framework or a container.

**DEPENDENCY INJECTION**

Dependency Injection is a design pattern that implements Inversion of Control by providing dependencies to objects from the outside, instead of letting objects create them internally.

- constructor injection

- method injection

- setter injection

**Built in Container IoC**

*builder.Services.AddTransient<IEmailService, EmailService>();*

*builder.Services.AddScoped<IUserService, UserService>();*

*builder.Services.AddSingleton<IConfigService, ConfigService>();*

**Singleton**  
A **single instance** is created and shared throughout the **application's life**.

### ✅ Use when:

* The service is **stateless**
* You want **cached/shared data**

Transient - New instance every time it’s requested : Lightweight, stateless services

Scoped - One instance per HTTP request : Business logic, services working with DB context

Singleton - One instance for the entire app :Logging, config providers, caching utilities

## Scoped

### ]

* A **new instance** is created for **each HTTP request**.

### ✅ Use when:

* You want data per-request (e.g. unit-of-work)
* Web-specific logic per user/session

services.AddScoped<IReportService, ReportService>();

## Transient

* A **new instance** is created **every time** it's requested.

### ✅ Use when:

* You want a fresh object each time
* The object holds **no state**

services.AddTransient<IReportService, ReportService>();

| **Feature** | **Use Case** | **Lifetime** |
| --- | --- | --- |
| **Constructor DI** | Mandatory dependency | Always available |
| **Setter DI** | Optional dependency | Late-binding allowed |
| **Method DI** | Per-method flexibility | Stateless usage |
| **Singleton** | One app-wide instance | Caching, config |
| **Scoped** | One per HTTP request | Web app middle layers |
| **Transient** | One per call | Lightweight, stateless |

## What is Middleware?

Middleware is a **software component** in the **HTTP request pipeline** that:

1. **Receives** an incoming HTTP request
2. **Performs some logic** (e.g., logging, authentication, routing, etc.)
3. **Decides to pass** the request to the next middleware in the pipeline — or stop it

### 🔁 Middleware Pipeline

The request flows through a **sequence of middleware** like this:

[Request] → [Middleware1] → [Middleware2] → [Middleware3] → [Controller]

↑

[Response] ← [Middleware3] ← [Middleware2] ← [Middleware1] ←

Each middleware can:

* **Inspect/modify** the request
* **Do work before/after** the next middleware runs
* **Short-circuit** the pipeline (e.g., return a response early)

## 🧠 Real-Life Analogy

Think of middleware like airport security checks 🛂 — each checkpoint inspects your baggage or documents and decides whether to let you through, modify your path, or stop you.

## 🧩 Common Middleware in ASP.NET Core

| **Middleware** | **What it does** |
| --- | --- |
| UseRouting() | Matches incoming URL to controller/action |
| UseAuthorization() | Verifies user access |
| UseAuthentication() | Authenticates user identity |
| UseExceptionHandler() | Catches unhandled exceptions |
| UseCors() | Handles cross-origin requests |
| UseHttpsRedirection() | Redirects HTTP to HTTPS |
| UseStaticFiles() | Serves static files |
| UseSwagger() | Enables API docs |

## 💻 Example Middleware in Program.cs

var app = builder.Build();

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapControllers();

app.Run();

## 🔨 Writing Custom Middleware

public class LoggerMiddleware

{

private readonly RequestDelegate \_next;

public LoggerMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

Console.WriteLine($"Request: {context.Request.Path}");

await \_next(context); // Call next middleware

Console.WriteLine($"Response: {context.Response.StatusCode}");

}

}

**Register it** in Program.cs:

app.UseMiddleware<LoggerMiddleware>();

## 🎯 Why Use Middleware?

* 🔍 Cross-cutting concerns (logging, security, exception handling)
* 🔁 Reusable and composable logic
* 🔄 Request/response inspection
* 🧱 Foundation of the ASP.NET Core pipeline

**JWT Authentication**

A **JWT** is a compact, URL-safe string used to securely transmit information between parties. It contains **claims** that are digitally signed using a secret key.

A **JWT (JSON Web Token)** is a **compact**, **URL-safe**, and **self-contained** token used to securely **transmit identity and claims** between parties. It’s an **open standard** (RFC 7519) used for **authentication and authorization** in modern applications.

## **What are Claims?**

**Claims** are **key-value pairs** included in the **payload** section of a JWT. They carry information about the **user** and the **token’s context**.

Example:

{

"sub": "admin",

"role": "Admin",

"email": "admin@example.com",

"exp": 1752146424

}

These claims are used to make **authorization decisions** on the server.

## **Why Use Claims?**

Claims help you:

* **Identify** the user (sub, name)
* **Authorize** based on role or permissions (role, scope)
* **Validate** the token (exp, iss, aud)
* **Avoid server-side session storage** (stateless authentication)

## **How Claims Work (Internally)**

1. **Server generates JWT** with claims after successful login.
2. The claims are **encoded into the JWT** (as the payload).
3. The client sends the JWT in the **Authorization header**.
4. On every request:
   * ASP.NET Core **verifies** the token and **extracts claims**
   * Claims are available in HttpContext.User.Claims

## **When Are Claims Used?**

* During **login**, the server inserts claims into the JWT.
* On **each API request**, claims are extracted and checked.
* When you protect routes like this:

[Authorize(Roles = "Admin")]

It checks the role claim in the token.

## **Where Are Claims Used?**

### 1. In the JWT payload:

{

"sub": "12345",

"role": "Admin",

"email": "user@example.com"

}

### 2. In code (e.g., AuthController):

var username = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

var role = User.FindFirst(ClaimTypes.Role)?.Value;

## **Types of Claims**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| **Registered** | Standard JWT claims | sub, exp, iss, aud, iat |
| **Public** | Defined by your app | email, username, role |
| **Private** | Internal use only | internalUserId, app-specific-data |

## **Use Cases of Claims**

|  |  |  |
| --- | --- | --- |
| **Use Case** | **Claim** | **Example** |
| Identify user | sub | "sub": "admin" |
| Role-based access | role | "role": "Admin" |
| Show email in UI | email | "email": "admin@example.com" |
| Expiration check | exp | "exp": 1752146424 |
| Multi-tenant apps | tenantId (custom) | "tenantId": "abc123" |

## Example: Use Claims in ASP.NET Controller

[Authorize]

[HttpGet("me")]

public IActionResult GetMe()

{

var username = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

var role = User.FindFirst(ClaimTypes.Role)?.Value;

return Ok(new { username, role });

}

## Summary: JWT Claims

|  |  |
| --- | --- |
| **Aspect** | **Summary** |
| What | Info embedded in JWT |
| Why | Used for authentication/authorization |
| How | Included by server, verified by middleware |
| When | On login (create), every request (read) |
| Where | JWT payload and HttpContext.User.Claims |

## JWT Structure

A JWT consists of **three parts**, separated by dots (.):

HEADER.PAYLOAD.SIGNATURE

### 1. **Header** (Base64-encoded JSON)

Specifies the token type and signing algorithm:

{

"alg": "HS256",

"typ": "JWT"

}

### 2. **Payload** (Base64-encoded JSON)

Contains **claims** – information about the user:

{

"sub": "admin",

"role": "Admin",

"exp": 1752146424,

"iss": "JwtAuthDemo",

"aud": "JwtAuthDemoUser"

}

Common claims:

* sub: subject (user ID or username)
* role: user’s role
* exp: expiration timestamp
* iss: issuer
* aud: audience

### 3. **Signature**

A **hashed combination** of:

Base64UrlEncode(header) + "." + Base64UrlEncode(payload)

Signed using the **secret key** and algorithm (e.g., HS256):

HMACSHA256(

base64UrlEncode(header) + "." + base64UrlEncode(payload),

secret

)

This makes the token **tamper-proof**.

## **JWT Authentication Flow**

### 1. **Login**

* User sends username and password.
* Server verifies credentials (e.g., using MongoDB + BCrypt).
* Server generates a JWT and returns it to the client.

### 2. **Client Stores Token**

* The client (e.g., browser or Angular app) stores the token (usually in localStorage or sessionStorage).

### 3. **Token in Request**

* For subsequent requests, the client sends:

Authorization: Bearer <token>

### 4. **Server Verifies JWT**

* ASP.NET reads the token via the JwtBearerMiddleware.
* It:
  + Checks the signature (with the secret key).
  + Validates iss, aud, exp, etc.
  + Parses claims and sets the HttpContext.User.

### 5. **Access Controlled by Claims**

* You can secure endpoints with:

[Authorize(Roles = "Admin")]

## Expired or Invalid Token

If the token is:

* Expired → 401 Unauthorized
* Tampered → Signature invalid → 401 Unauthorized.

## Why It's Secure

* **Tamper-proof**: Signature ensures the payload wasn’t modified.
* **Stateless**: Server doesn’t store sessions. All info is in the token.
* **Self-contained**: Includes all user data required to verify identity.

## **WHY is JWT Used?**

### Benefits:

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| **Stateless** | Server doesn't need to store sessions. |
| **Compact** | Efficient for HTTP headers, mobile, cookies. |
| **Self-contained** | Includes user identity and permissions. |
| **Cross-platform** | Works with any client (Angular, mobile, etc.). |
| **Scalable** | Ideal for microservices and APIs. |

## **WHERE is JWT Used?**

### Common Places:

* Web APIs with Angular, React, Vue
* Mobile apps (iOS, Android)
* Microservices and gateways (e.g., API Gateway)
* OAuth 2.0 / OpenID Connect implementations (Auth0, Azure AD)

## **HOW Does JWT Work (Internally)?**

### 1. User Logs In

* Sends credentials to POST /api/login

### 2. Server Verifies & Issues JWT

* If valid, server creates a JWT:
  + **Header**: algorithm (e.g., HS256)
  + **Payload**: user claims (e.g., sub, role)
  + **Signature**: signed using secret/private key

### 3. Client Stores JWT

* In localStorage, sessionStorage, or cookies

### 4. Client Sends JWT on Requests

* In HTTP Header:

Authorization: Bearer <jwt-token>

### 5. Server Verifies JWT

* Middleware decodes and verifies signature, expiry, issuer, etc.

### 6. Authorization Checks

* Use [Authorize] or [Authorize(Roles="Admin")] to restrict access based on claims.

## **WHEN to Use JWT**

### Best Use Cases:

* Stateless authentication (REST APIs)
* Role-based access control (RBAC)
* Microservices
* SSO (Single Sign-On)
* OAuth/OpenID Connect identity propagation

### When **Not** to Use:

* Server-side session tracking (traditional web apps)
* Short-lived tokens with frequent revocation needs (unless combined with refresh tokens)

## **INTERNAL WORKINGS: Deep Dive**

### Token Creation:

new JwtSecurityTokenHandler().WriteToken(new JwtSecurityToken(...));

Creates:

* Encoded header and payload
* Signs with HMAC or RSA
* Returns a base64 string

### Token Verification:

* Signature is validated using secret key (HMAC) or public key (RSA)
* Expiry is checked via exp claim
* Claims are extracted and injected into HttpContext.User

## JWT in Industry-Grade Setup

|  |  |
| --- | --- |
| **Feature** | **Implementation** |
| **Signing** | HMAC for simplicity, RSA for security |
| **Rotation** | Rotate secret keys every X days |
| **Refresh Tokens** | Short-lived access tokens (e.g. 15 mins), long-lived refresh tokens |
| **Secure Storage** | HttpOnly cookies or secure localStorage |
| **Revocation** | Token blacklist (DB or Redis), or use short TTL |
| **Claims Mapping** | Map roles, permissions, tenantId, etc. |

## Real-World Example

**Login:**

POST /api/auth/login

{

"username": "admin",

"password": "password"

}

**Response:**

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp..."

}

**Subsequent Call:**

GET /api/user/me

Authorization: Bearer eyJhbGciOiJIUzI1NiIs...

## Summary: JWT in Industrial Standards

|  |  |
| --- | --- |
| **Aspect** | **Description** |
| What | A secure token with user claims |
| Why | Stateless, scalable, efficient |
| Where | APIs, mobile, SSO, microservices |
| How | Issued on login, sent via headers, validated by middleware |
| When | Ideal for REST APIs, OAuth flows |
| Internals | Header + Payload + Signature; validated using secret/private key |

## 1. **Create ASP.NET Core Web API Project**

### Using Visual Studio:

* Open Visual Studio
* Select **Create a new project**
* Choose **ASP.NET Core Web API**
* Name the project (e.g., JwtAuthDemo)
* Set target framework to **.NET 8**
* Uncheck "Use controllers (minimal API)" if you want full controller support
* Click **Create**

## 2. **Install Required NuGet Package**

Install the JWT Bearer authentication package:

Install-Package Microsoft.AspNetCore.Authentication.JwtBearer

Or via CLI:

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

## 3. **Configure JWT Authentication in** Program.cs

### Add Configuration:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

// JWT Authentication Configuration

builder.Services.AddAuthentication("Bearer")

.AddJwtBearer("Bearer", options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = builder.Configuration["Jwt:Issuer"],

ValidAudience = builder.Configuration["Jwt:Audience"],

IssuerSigningKey = new SymmetricSecurityKey(

Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))

};

});

builder.Services.AddAuthorization();

builder.Services.AddCors(options =>

{

options.AddPolicy("AllowAll", policy =>

{

policy.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader();

});

});

var app = builder.Build();

app.UseCors("AllowAll");

app.UseAuthentication();

app.UseAuthorization();

app.MapControllers();

app.Run();

## 4. **Add JWT Configuration in** appsettings.json

"Jwt": {

"Key": "ThisIsASecretKeyForJwtTokenDontShare",

"Issuer": "JwtAuthDemo",

"Audience": "JwtAuthDemoUser"

}

## 5. **Create Auth Controller (**Controllers/AuthController.cs**)**

### Define Login Model

public class LoginModel

{

public string Username { get; set; }

public string Password { get; set; }

}

### Create Controller

[ApiController]

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

public class AuthController : ControllerBase

{

private readonly IConfiguration \_config;

public AuthController(IConfiguration config)

{

\_config = config;

}

[HttpPost("login")]

public IActionResult Login([FromBody] LoginModel login)

{

if (login.Username == "admin" && login.Password == "password")

{

var token = GenerateJwtToken(login.Username);

return Ok(new { token });

}

return Unauthorized();

}

private string GenerateJwtToken(string username)

{

var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_config["Jwt:Key"]));

var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

var claims = new[]

{

new Claim(JwtRegisteredClaimNames.Sub, username),

new Claim(ClaimTypes.Role, "Admin"),

new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())

};

var token = new JwtSecurityToken(

issuer: \_config["Jwt:Issuer"],

audience: \_config["Jwt:Audience"],

claims: claims,

expires: DateTime.Now.AddMinutes(30),

signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);

}

}

## 6. **Protect Your APIs**

Use [Authorize] to protect endpoints:

[Authorize]

[HttpGet("secure")]

public IActionResult SecureData()

{

return Ok("This is protected data");

}

To allow specific roles:

[Authorize(Roles = "Admin")]

## 7. **Test Using Postman**

1. **POST** to /api/auth/login with:

{

"username": "admin",

"password": "password"

}

1. Use the received **JWT token** in Authorization header as:

Authorization: Bearer <token>

1. **GET** /api/auth/secure with token.

**Exception Handling**

**Global Exception Handling via Middleware**

This is **framework-agnostic** and works across all request pipelines (MVC, Razor Pages, minimal APIs, etc.).

### ****Best Practices****

1. **Catch only what you can handle.** Don’t use general catch (Exception) unless necessary.
2. **Use finally** for cleanup (like closing DB connections or streams).
3. **Avoid swallowing exceptions** (i.e., catching and doing nothing).
4. **Log exceptions** for diagnostics (e.g., using Serilog or NLog).
5. **Throw new or rethrow correctly:**

Middleware is the **perfect place** to centralize error handling for your app. This way, any unhandled exceptions thrown in your controllers, services, etc., are caught and dealt with in **one place**.

## Custom Exception Handling Middleware in ASP.NET Core

### Step-by-step plan:

1. Create a custom middleware.
2. Use try-catch around the next() delegate.
3. Return structured error response (often JSON).
4. Register the middleware in the request pipeline.

### Example: ExceptionHandlingMiddleware.cs

public class ExceptionHandlingMiddleware

{

private readonly RequestDelegate \_next;

private readonly ILogger<ExceptionHandlingMiddleware> \_logger;

public ExceptionHandlingMiddleware(RequestDelegate next, ILogger<ExceptionHandlingMiddleware> logger)

{

\_next = next;

\_logger = logger;

}

public async Task InvokeAsync(HttpContext context)

{

try

{

await \_next(context); // continue processing

}

catch (Exception ex)

{

\_logger.LogError(ex, "Unhandled exception occurred");

context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;

context.Response.ContentType = "application/json";

var response = new

{

error = ex.Message,

stackTrace = ex.StackTrace,

statusCode = context.Response.StatusCode

};

var json = JsonSerializer.Serialize(response);

await context.Response.WriteAsync(json);

}

}

}

### Register the Middleware in Program.cs (or Startup.cs)

app.UseMiddleware<ExceptionHandlingMiddleware>();

Place it **early** in the middleware pipeline, usually just after UseRouting() but before UseEndpoints().

### Bonus: Custom Exceptions

You can also check the type of exception and return different status codes:

catch (Exception ex)

{

int statusCode = ex switch

{

UnauthorizedAccessException => StatusCodes.Status401Unauthorized,

KeyNotFoundException => StatusCodes.Status404NotFound,

\_ => StatusCodes.Status500InternalServerError

};

context.Response.StatusCode = statusCode;

}

## What is Middleware?

In ASP.NET Core, **middleware** is like a **pipeline** through which all HTTP requests pass.

Imagine this:

[User Request] --> [Middleware1] --> [Middleware2] --> [Your Controller Code]

Middleware can inspect, change, or stop the request/response at any point.

## Problem: What if your code throws an error?

public IActionResult GetData()

{

throw new Exception("Something broke!");

}

If you don't catch this, the app crashes or shows an ugly error. So we build **middleware** to catch it.

## Easy Example: Exception Middleware

### 1. Create a Class to Catch Errors

public class ErrorMiddleware

{

private readonly RequestDelegate \_next;

public ErrorMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task Invoke(HttpContext context)

{

try

{

await \_next(context); // Call the next middleware/controller

}

catch (Exception ex)

{

// Handle the error

context.Response.StatusCode = 500;

context.Response.ContentType = "text/plain";

await context.Response.WriteAsync("Something went wrong: " + ex.Message);

}

}

}

### 2. Register It in Program.cs

In your Program.cs file, add:

app.UseMiddleware<ErrorMiddleware>();

This tells the app: “Before doing anything else, run my error handler.”

## Result:

If **any** error happens, your middleware catches it and sends a nice message instead of crashing the app.

## What's an Exception Filter?

An **Exception Filter** is a special class that catches unhandled exceptions thrown by controllers, and lets you decide what to do — log it, change the HTTP status code, return custom JSON, etc.

## Step-by-Step: Global Exception Filter in ASP.NET Core

### 1. Create the Exception Filter

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

using System.Net;

public class GlobalExceptionFilter : IExceptionFilter

{

private readonly ILogger<GlobalExceptionFilter> \_logger;

public GlobalExceptionFilter(ILogger<GlobalExceptionFilter> logger)

{

\_logger = logger;

}

public void OnException(ExceptionContext context)

{

\_logger.LogError(context.Exception, "Unhandled exception caught by global filter");

var response = new

{

message = "An unexpected error occurred",

details = context.Exception.Message

};

context.Result = new JsonResult(response)

{

StatusCode = (int)HttpStatusCode.InternalServerError

};

context.ExceptionHandled = true; // Marks the exception as handled

}

}

### 2. Register It Globally

#### For ASP.NET Core 6+ in Program.cs:

builder.Services.AddControllers(options =>

{

options.Filters.Add<GlobalExceptionFilter>();

});

This ensures the filter runs for **every controller action**.

### 3. Try Throwing an Exception

[ApiController]

[Route("[controller]")]

public class TestController : ControllerBase

{

[HttpGet]

public IActionResult Boom()

{

throw new Exception("💥 Controller blew up!");

}

}

You’ll now get a clean JSON response like:

{

"message": "An unexpected error occurred",

"details": "💥 Controller blew up!"

}

## Optional Upgrade: Customize StatusCode Based on Exception

You can add logic like:

int statusCode = context.Exception switch

{

ArgumentException => 400,

UnauthorizedAccessException => 401,

\_ => 500

};

And use it in StatusCode.

## Step-by-Step Guide

### 1. Create a New Filter Class

This class will handle the exceptions globally:

### 2. Tell ASP.NET to Use This Filter Globally

In your Program.cs (if you're using .NET 6 or later), add this:

### 3. Try It with a Controller

Create this test controller:

[ApiController]

[Route("[controller]")]

public class CrashController : ControllerBase

{

[HttpGet]

public IActionResult Test()

{

throw new Exception("🔥 Oops, something exploded!");

}

}

Now run the app and visit /crash.

Instead of a server error, you'll get this clean JSON response:

{

"message": "An error occurred",

"error": "🔥 Oops, something exploded!"

}

Why Is This Cool?

Catches all controller exceptions

Sends friendly messages to the user

Keeps your code DRY (no repeated try/catch)

Super easy to extend later (custom exceptions, status codes, logging)

# Middlewares in ASP.NET Core

Exception handling forms the bedrock of robust web applications. In this blog post, we’ll take a humble approach to tackle this essential aspect of ASP.NET Core development. Our focus is on building a basic understanding, laying the groundwork for more complex exception handling strategies.

We’ll start from scratch, creating a clean ASP.NET Core Web API project. With simplicity in mind, I’ll guide you through setting up a custom ExceptionMiddleware. This fundamental middleware will catch errors globally, providing a stable foundation upon which you can construct more intricate exception handling mechanisms in the future.

**Custom Exception Handling Middleware**

Now, let’s delve into the custom middleware responsible for handling exceptions globally. In the Middlewares folder, create a file named ExceptionMiddleware.cs. This middleware intercepts exceptions, logs them, and sends a JSON-formatted error response to the client.

using ExceptionHandlingProject.Models;  
using System.Net;  
using System.Net.Mime;  
using System.Text.Json;  
  
namespace ExceptionHandlingProject.Extensions  
{  
 public class ExceptionMiddleware  
 {  
 private readonly ILogger<ExceptionMiddleware> \_logger;  
 private readonly RequestDelegate \_next;  
  
 public ExceptionMiddleware(ILogger<ExceptionMiddleware> logger, RequestDelegate next)  
 {  
 \_logger = logger;  
 \_next = next;  
 }  
  
 public async Task InvokeAsync(HttpContext context)  
 {  
 try  
 {  
 await \_next(context);  
 }  
 catch (Exception ex)  
 {  
 \_logger.LogError(ex, ex.Message);  
 await HandleCustomExceptionResponseAsync(context, ex);  
 }  
 }  
  
 private async Task HandleCustomExceptionResponseAsync(HttpContext context, Exception ex)  
 {  
 context.Response.ContentType = MediaTypeNames.Application.Json;  
 context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;  
  
 var response = new ErrorModel(context.Response.StatusCode, ex.Message, ex.StackTrace?.ToString());  
 var options = new JsonSerializerOptions { PropertyNamingPolicy = JsonNamingPolicy.CamelCase };  
   
 var json = JsonSerializer.Serialize(response, options);  
 await context.Response.WriteAsync(json);  
 }  
 }  
}

Let’s break down our custom middleware’s functionality step by step:

**Middleware Class Definition:**

public class ExceptionMiddleware  
{  
 private readonly ILogger<ExceptionMiddleware> \_logger;  
 private readonly RequestDelegate \_next;  
  
 public ExceptionMiddleware(ILogger<ExceptionMiddleware> logger, RequestDelegate next)  
 {  
 \_logger = logger;  
 \_next = next;  
 }

The ExceptionMiddleware class is defined, which takes ILogger<ExceptionMiddleware> and RequestDelegate as constructor parameters. ILogger is used for logging exceptions, and RequestDelegate represents the next middleware in the pipeline.

**InvokeAsync Method:**

public async Task InvokeAsync(HttpContext context)  
{  
 try  
 {  
 await \_next(context);  
 }  
 catch (Exception ex)  
 {  
 \_logger.LogError(ex, ex.Message);  
 await HandleCustomExceptionResponseAsync(context, ex);  
 }  
}

The InvokeAsync method is the entry point of the middleware. It catches exceptions occurring during the execution of subsequent middlewares or the request handling pipeline.

* In the try block, it attempts to execute the next middleware by invoking \_next(context).
* If an exception is thrown, it’s caught in the catch block. The exception is logged using the provided logger (\_logger), and HandleCustomExceptionResponseAsync method is invoked to manage the exception and send a tailored error response.

**HandleCustomExceptionResponseAsync Method:**

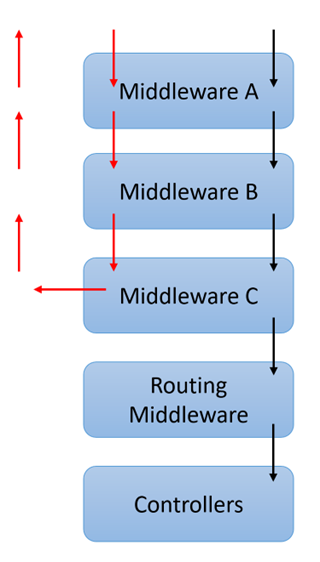
private async Task HandleCustomExceptionResponseAsync(HttpContext context, Exception ex)  
{  
 context.Response.ContentType = MediaTypeNames.Application.Json;  
 context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;  
  
 var response = new ErrorModel(context.Response.StatusCode, ex.Message, ex.StackTrace?.ToString());  
 var options = new JsonSerializerOptions { PropertyNamingPolicy = JsonNamingPolicy.CamelCase };  
   
 var json = JsonSerializer.Serialize(response, options);  
 await context.Response.WriteAsync(json);  
}

This method is responsible for crafting and transmitting the custom error response.

* It sets the response content type to JSON and the status code to 500 (Internal Server Error).
* It creates an ErrorModel object, passing the status code, exception message, and stack trace (if available).
* The ErrorModel is serialized to JSON using JsonSerializer and sent as the HTTP response.

**Filters vs. Middlewares**

*Filters* in ASP.NET Core are components that can be applied globally, at the controller level, or on specific action methods. They allow us to execute logic before or after the execution of an action method. Exception filters, specifically, enable the handling of exceptions that occur during the execution of controller actions. By using filters, we can encapsulate exception handling logic directly within the controller or action context, making it a more granular and specific approach.

Here is a good example that shows the difference more clearly:

The arrows represent the flow of an HTTP request.

Imagine the flow of an HTTP request as a sequence of [Matryoshka dolls](https://cdn2.bigcommerce.com/server5500/ammn9sj/products/575/images/27128/ND7617__63656.1670706172.1280.1280.jpg?c=2) nested within one another. The flow begins with middlewares — these are like the outermost dolls, capturing requests at the earliest stage. As the request progresses, it passes through various middlewares, each encapsulating a specific behavior. When the routing middleware steps in, the request starts to traverse the MVC pipeline, eventually reaching the controllers.

**When to Use Middlewares**

Middlewares are your go-to choice when you need to manage the flow and execution of requests independently of the MVC context. Whether you’re handling headers, implementing pre-routing mechanisms, or orchestrating intricate request-response patterns, middlewares offer the flexibility and broad scope necessary for these tasks.

**When to Opt for Filters**

On the other hand, filters come into play when you require the context of MVC, particularly if your focus is on actions within controllers. Filters operate at a more granular level, enabling you to inject specific logic before or after the execution of action methods. If your exception handling or any other operation directly relates to MVC components, filters provide the targeted approach you need.

In simple terms, middlewares oversee the overall journey of requests throughout the entire application, making them suitable for tasks that cover the entire flow. Filters, on the other hand, focus on specific parts of the application like actions in controllers, allowing for more targeted enhancements. Your choice between them depends on what your application needs, ensuring you use the right tool for the specific task at hand.

**Implementing the Execution Filter**

Remember our Error Model? We are going to keep using it for our exception filter’s response body too. Let’s remember the error model again:

namespace ExceptionHandlingProject.Models  
{  
 public class ErrorModel  
 {  
 public int StatusCode { get; set; }  
 public string? Message { get; set; }  
 public string? Details { get; set; }   
  
 public ErrorModel(int statusCode, string? message, string? details = null)  
 {  
 StatusCode = statusCode;  
 Message = message;  
 Details = details;  
 }  
 }  
}

Now create a folder called Filters in the project root folder and create a file named ExceptionFilter.csin there.

using ExceptionHandlingProject.Models;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.AspNetCore.Mvc.Filters;  
  
namespace ExceptionHandlingProject.Filters  
{  
 public class ExceptionFilter : IExceptionFilter  
 {  
 public void OnException(ExceptionContext context)  
 {  
 var error = new ErrorModel  
 (  
 500,  
 context.Exception.Message,  
 context.Exception.StackTrace?.ToString()  
 );  
  
 context.Result = new JsonResult( error );  
 }  
 }  
}

By implementing the IExceptionFilter interface, this class gains the ability to handle exceptions that occur within MVC action methods.

The OnException method is part of the IExceptionFilter interface and is invoked whenever an exception occurs within an action method. Inside this method:

* The ExceptionContext object provides access to the context of the exception.
* A new ErrorModel object is created, encapsulating the HTTP status code (500 for Internal Server Error), exception message, and stack trace (if available).
* The context.Result property is set to a new JsonResult containing the ErrorModel. This ensures that the error information is returned as JSON in the response.

Our filter is complete. Now let’s go to our HomeController.csand add the following attribute to the Get() method.

using ExceptionHandlingProject.Filters;  
using Microsoft.AspNetCore.Mvc;  
  
namespace ExceptionHandlingProject.Controllers  
{  
 [Route("api/[controller]")]  
 [ApiController]  
 public class HomeController : ControllerBase  
 {  
 [HttpGet]  
 [ServiceFilter(typeof(ExceptionFilter))] //Add this attribute here  
 public IActionResult Get()  
 {  
 throw new Exception("Exception in HomeController.");  
 }  
 }  
}

By attaching [ServiceFilter(typeof(ExceptionFilter))] to the Get method, we are instructing ASP.NET Core to apply the ExceptionFilter globally to this specific action. This means any exceptions thrown during the execution of the Get method will be caught and processed by the ExceptionFilter before the response is sent back to the client.

Finally, let’s register our filter to the pipeline in Program.cs . Add the following line before the Services.AddControllers() method is called and comment the app.UseMiddleware<ExceptionMiddleware>(); line to test our filter.

builder.Services.AddScoped<ExceptionFilter>();

This line registers the ExceptionFilter as a service in ASP.NET Core's dependency injection container. This registration allows the ExceptionFilter to be used globally within ourapplication. By applying the [ServiceFilter(typeof(ExceptionFilter))] attribute to specific actions in controllers, ASP.NET Core automatically resolves and applies the ExceptionFilter to those actions, ensuring consistent and centralized exception handling at action level.

**CQRS**

CQRS is a pattern that suggests separating the "command" (write operations) from the "query" (read operations). This leads to better scalability, maintainability, and clarity in your codebase.

Commands → actions that change state (CreateProduct, UpdateOrder)

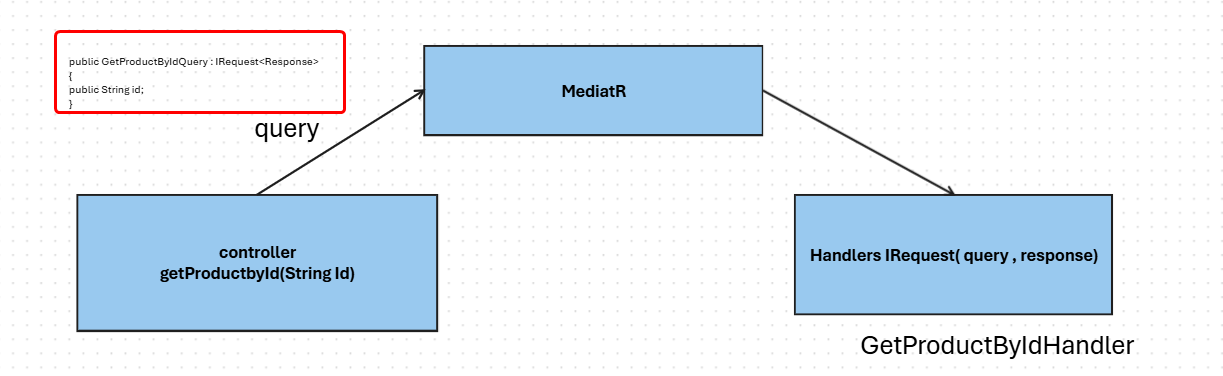
Queries → actions that read data (GetProductById, GetOrdersList)

In the CQRS pattern, MediatR will handle the communication between the client code (controller) and the handler (command/query).

- The controller will send a command or query through the IMediator interface.

- MediatR will then map the command/query to the appropriate handler and execute it

**MediatR**



MediatR is a .NET library that implements the Mediator Pattern.

It lets you decouple the sender and receiver:

Instead of calling services directly from controllers (e.g., \_productService.Add()),

You send a command or query through MediatR:

*ProductController.cs :*

*GetProductById(String id)*

*{*

*res = await \_mediator.Send(new GetProductQueryByIdRequest { productId= id});*

*}*

**Step-by-Step Flow with MediatR:**

1. You Create a Request (Query or Command):

- A request is usually a simple class implementing IRequest<TResponse>, where TResponse is the type of data returned by the handler.

2. You Send the Request Using IMediator.Send(query):

- The controller or service sends the request using the IMediator interface.

- MediatR will then look for a matching handler for that specific type of request.

3 .MediatR Finds the Right Handler:

- MediatR internally uses dependency injection (DI) to resolve the correct handler.

- It uses the type of the request (TRequest) to locate a handler that implements IRequestHandler<TRequest, TResponse>.

4. The Handler Executes the Logic:

- Once the handler is found, MediatR invokes its Handle method, passing the request as a parameter.

- The handler performs the necessary logic and returns a response (e.g., a Car, List<Car>, bool, etc.).

5. MediatR Returns the Response:

- The response is then returned to the calling component (e.g., the controller), which can use it.

**CQRS pattern**

What is CQRS?, this acronym stands for **Command and Query Responsibility Segregation**, and this architectural pattern is used to separate the operations that read data from the operations that write or update data, let’s explore the problem this architecture aims to solve, in case you’re wondering why you should use it.

**The problem**

It is common for applications to perform a large number of **read/write** operations, however, when the user of the application and the quantity of the operations grows exponentially, it is likely that the application will suffer issues such as:

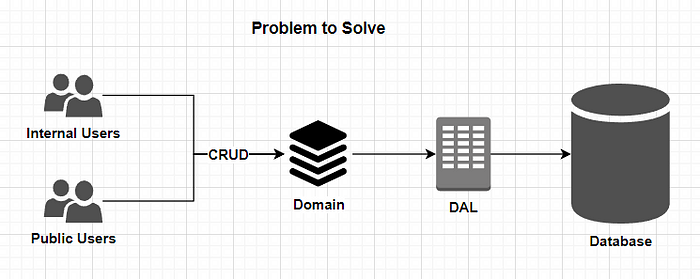
**Performance:** Read and write operations differ in their performance requirements and characteristics. Managing these operations in the same model can lead to suboptimal performance, as resources are not allocated efficiently based on each operation’s specific needs.

**Scalability:** There is usually a discrepancy between the number of read and write operations in an application. Certain parts of the application may require handling a significant number of reads, while others may have to manage a high volume of writes. In such cases, it can be inefficient and costly to scale the entire system to handle both types of operations equally.

**Flexibility:** In tightly coupled architectures where read and write operations are interconnected, making changes will probably cause problems that will cost time and money.

**Consistency:** Maintaining consistency between read and write operations within the same model can be challenging, as it can lead to data inconsistencies, race conditions, and concurrency issues, potentially compromising the integrity of the system.

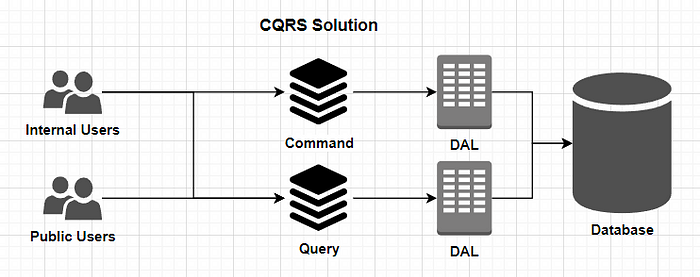
**Complexity:** When the application reads data, it may need to execute complex queries that produce Data Transfer Objects (DTOs) with different structures. This can result in complex object mapping. On the other hand, when writing data, the model may include complex validation and business logic, which can lead to an overly detailed model with multiple responsibilities.



Problem

**The Solution**

As previously mentioned, **CQRS**is a pattern that separates **queries**and **commands**within our application. This separation allows users to create and use different data models according to specific scenarios. Therefore, the application becomes more flexible, changes can now be made without breaking other modules, and there is no longer a need to lock tables or records for updates, which can negatively impact performance. Additionally, you can customize your data to fit your specific query needs, even denormalizing it if necessary. This means that you can use different storage technologies depending on the query requirements, and customize everything you need. Furthermore, you can scale and optimize read or write operations separately, leading to more efficient use of resources and less investment in infrastructure.



Solution

**Mediator pattern**

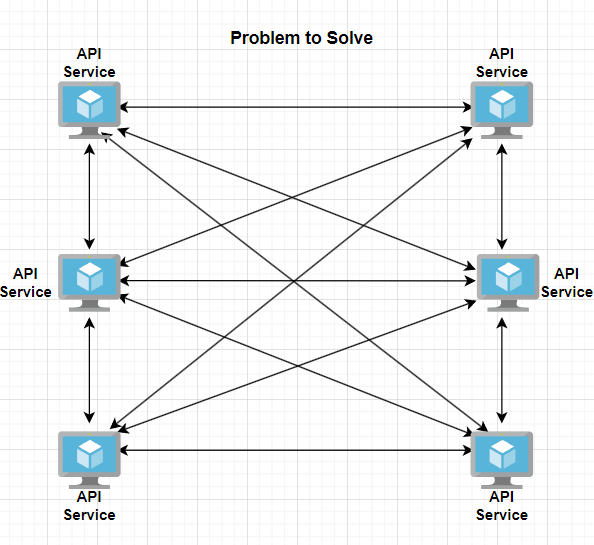
The mediator design pattern aims to reduce dependencies between objects by restricting direct communication and instead creating a way for them to collaborate only through the mediator object. In simpler terms, there is an object that encapsulates and manages how other objects interact with each other.

**The problem**

**Flexibility:** As business requirements evolve, it becomes increasingly important to have flexible applications that can adapt to changing needs. However, applications that are tightly coupled can make it difficult to make changes or add new features. A change in one object can have a ripple effect throughout the entire application, this can cause delays in development and increase the risk of introducing bugs or other issues.

**Complex management:** Each object should be aware of the interfaces and implementations of other objects it interacts with. However, this can make the system more complicated to comprehend, maintain, and develop and this issue grows over time like a snowball, directly impacting the developers, management, and the final product used by the clients.

**Coupling:**Objects that communicate directly are usually tightly coupled, in these cases making changes to those objects could impact others, leading to more changes in models, scripts, etc.



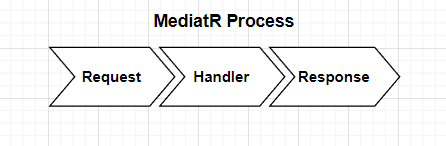
Problem

**The Solution**

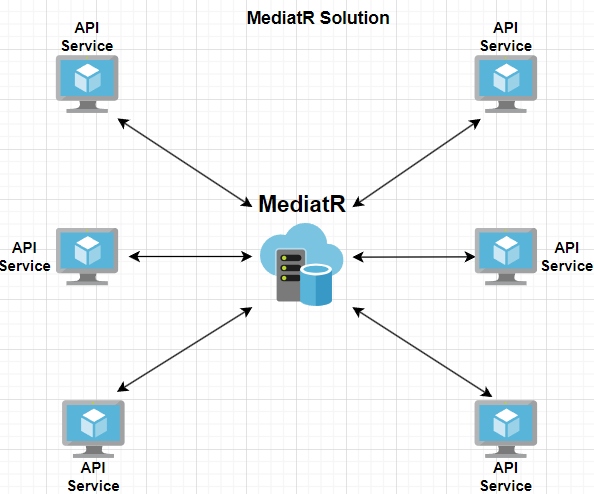
In the image below we can see how the **Mediator**pattern works, we have some services that can communicate with each other through a mediator object.

By allowing services to communicate without direct references, the application becomes more flexible, with simplified interactions and reduced dependencies, making it easier to maintain.

In this article, we are going to use the **Mediator**pattern through a library called **MediatR**. This library offers classes that allow efficient communication between multiple objects in a loosely coupled manner, simplifying the exchange of information, and making it easier to manage existing features and integrate new ones. It’s important to note that MediatR follows the Single Responsibility Principle and Dependency Inversion Principle.



Process



Solution

**Requirements**

* [Visual Studio (Community)](https://visualstudio.microsoft.com/es/vs/community/)
* [.Net 8](https://dotnet.microsoft.com/en-us/download/dotnet/8.0)
* ASP Web API project
* MediatR (library)

**NuGet package for Mediator**

The library **MediatR**offers the necessary functionality for implementing the Mediator pattern. You can obtain the library from NuGet packages in Visual Studio or via one of these methods:

**.NET CLI:**

dotnet add package MediatR --version 12.2.0

**Package manager console:**

Install-Package MediatR -Version 12.2.0

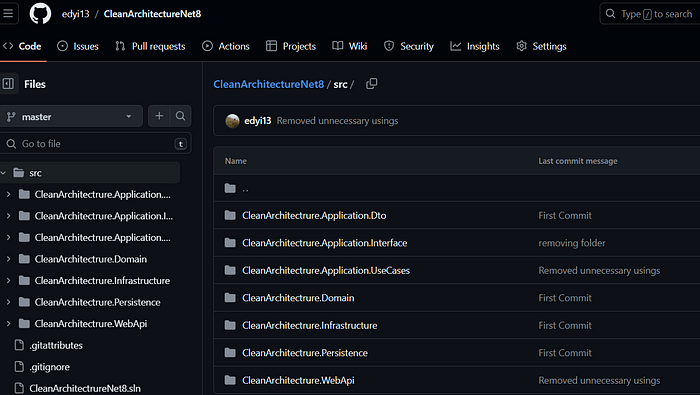
**Direct download from the NuGet website:**

[MediatR](https://www.nuget.org/packages/MediatR/)

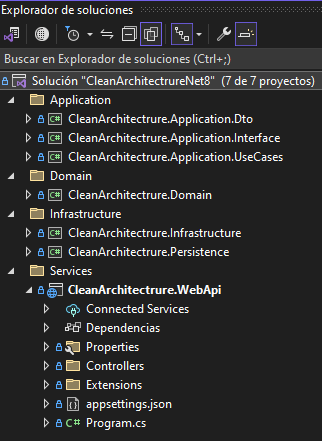
**Setting up the project**

As mentioned before you will need an API, in this case, I will use this API built using **Clean architecture**.

[Link to repository](https://github.com/edyi13/CleanArchitectureNet8)



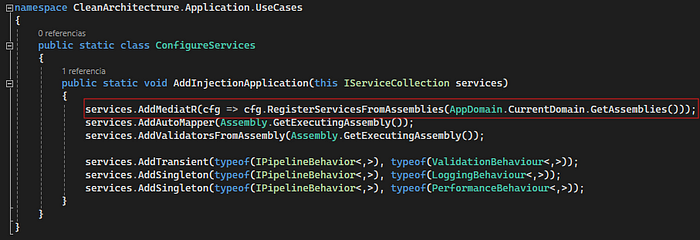
Github repo

**Project structure**

**Step 1:** Adding the MediatR dependency.

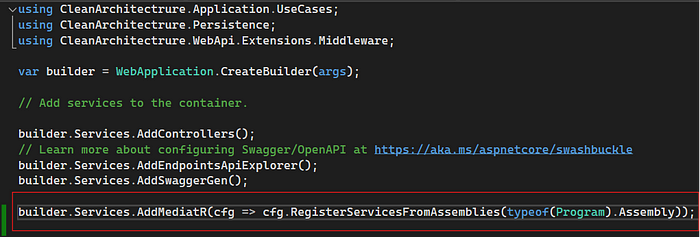
If you are using a custom class for adding dependencies use this option:

services.AddMediatR(cfg => cfg.RegisterServicesFromAssemblies(AppDomain.CurrentDomain.GetAssemblies()));



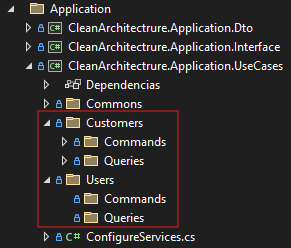
If not add it directly to the **Program.cs**:

builder.Services.AddMediatR(cfg => cfg.RegisterServicesFromAssemblies(typeof(Program).Assembly));

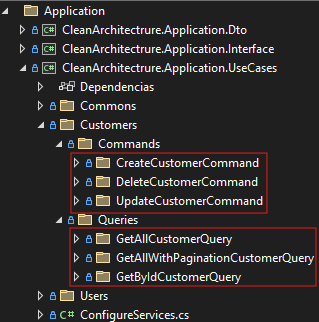


**Step 2:** Creating the folder structure

In the application layer we are going to create a folder for our entities and split our **Commands**and **Queries**into folders:



Inside the **Commands**and **Queries**folders, you are going to create a folder for each function you need to execute:



**Step 3:** Creating base classes

The classes required for using the CQRS and Mediator pattern are the Query/Command and Handler classes. The image below illustrates these classes:

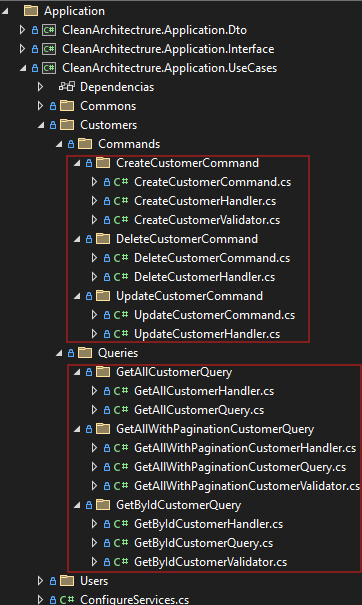
**Commands**

* CreateCustomerCommand.cs
* CreateCustomerHandler.cs
* CreateCustomerValidator.cs

**Queries**

* GetByIdCustomerQuery.cs
* GetByIdCustomerHandler.cs
* GetByIdCustomerValidator.cs

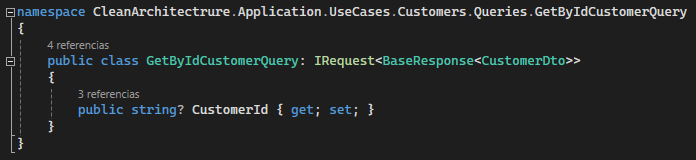
**Note:** The Validator class is optional, in this project the library used for validation is FluentValidation.



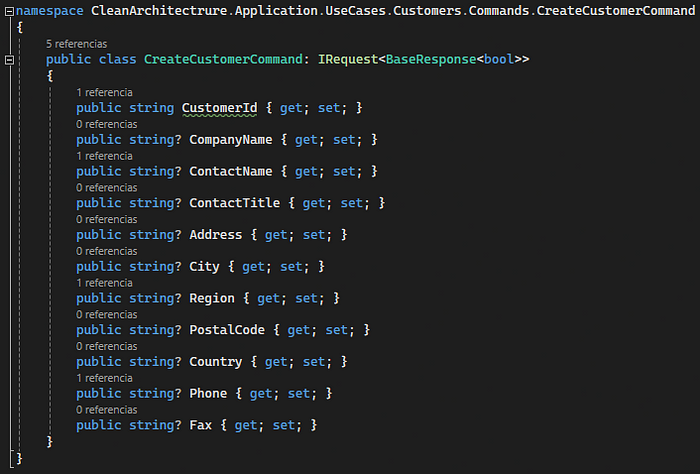
**Step 4:**Create the Query/Command class

First, we need to create our **Query/Command** class. In this class, we will inherit from the **IRequest**interface provided by **MediatR**. This interface informs **MediatR**that a request is being made and that it needs to be handled by the **Handler**class, you need to specify the type of request you are executing and pass it to IRequest.

**Note:** In this case, I have used a base response class to provide a generic response, but it is not mandatory.



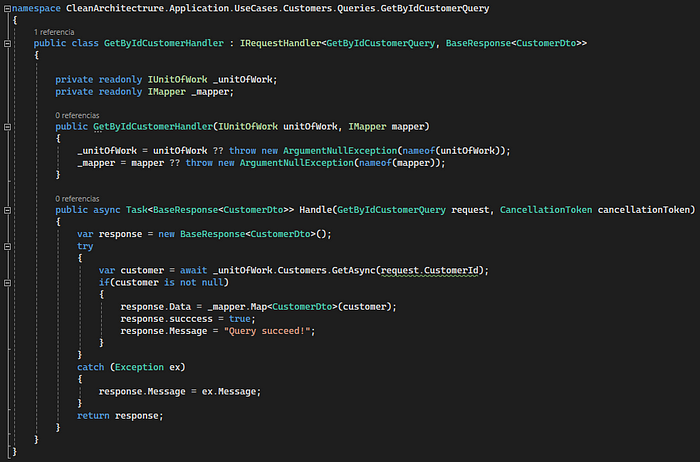
Query class



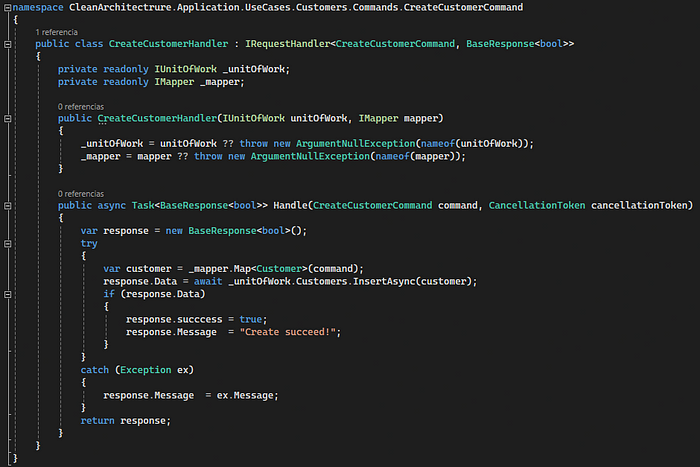
Command class

**Step 5:** Create the Handler class

This class will handle our requests based on whether it is a command or a query. For command requests, the class will update the application’s state as needed, while for query requests, the handler will fetch data based on the queries and return it to the client. In this class, we inherit the **IRequestHandler**, this interface takes the request (command/query) and the return type as parameters.



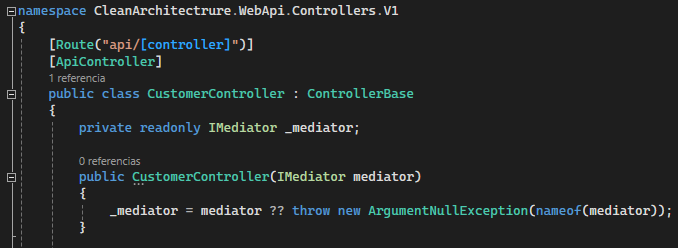
QueryHandler class



CommandHandler class

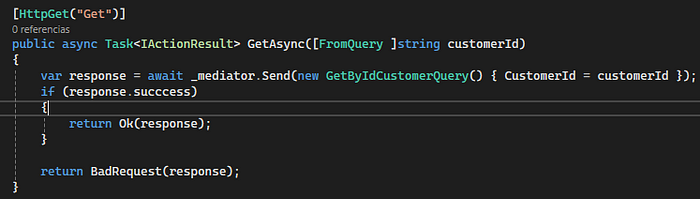
**Step 6:** Implementing MediatR in the Controller

To execute the commands and queries, you need to inject the **MediatR**library. Therefore, you must create a global read-only variable of the interface **IMediator**and initialize this variable using a constructor.

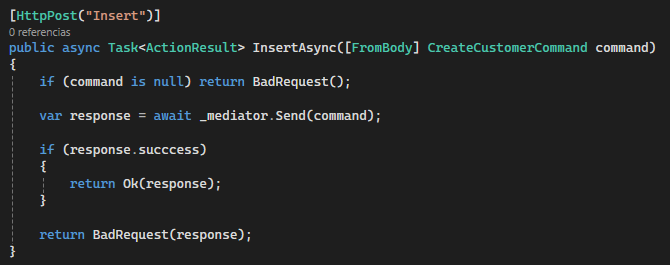


MediatR

After setting up the global read-only variable **\_mediator**, you can access the **Command/Query** functionalities through it. The **MediatR**library will handle the mapping of the requests to the appropriate handlers, ensuring that the commands and queries are processed correctly.



Query implementation



Command implementation

**Conclusion**

We have covered the essentials of implementing the CQRS pattern and the Mediator pattern in a .NET 8 Web API. Through this exploration, we have discovered that these two technologies can be valuable allies in improving our applications.

## \_mediator.Send(...) → **CQRS Style Request/Response**

This is the most commonly used method.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Request-Response** | One request gets **one handler**, and returns a response. |
| Used for | **Commands** and **Queries** in CQRS |
| Interface | IRequest<TResponse> + IRequestHandler<TRequest, TResponse> |
| Testable | Yes, because it's tightly controlled & isolated |

### Example

public class GetCarByIdQuery : IRequest<Car>

{

public string Id { get; set; }

}

public class GetCarByIdHandler : IRequestHandler<GetCarByIdQuery, Car>

{

public Task<Car> Handle(GetCarByIdQuery request, CancellationToken ct)

{

var car = \_repo.Get(request.Id);

return Task.FromResult(car);

}

}

var car = await \_mediator.Send(new GetCarByIdQuery { Id = "123" });

Think of Send() like a **command/query dispatcher**.

## mediator.Publish(...) → **Event Broadcasting / Notifications**

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **One-to-Many** | A notification is **broadcast to many handlers** |
| Fire-and-Forget | No response expected |
| Used for | **Domain Events, Notifications**, Logging, Audit, etc. |
| Side-effects | You can trigger logging, emails, alerts... from multiple places |

### Interface

* Request: INotification
* Handler: INotificationHandler<TNotification>

### Example

public class CarCreatedNotification : INotification

{

public string CarId { get; set; }

}

public class SendWelcomeEmailHandler : INotificationHandler<CarCreatedNotification>

{

public Task Handle(CarCreatedNotification notification, CancellationToken ct)

{

Console.WriteLine($"Send email for Car ID: {notification.CarId}");

return Task.CompletedTask;

}

}

public class LogToConsoleHandler : INotificationHandler<CarCreatedNotification>

{

public Task Handle(CarCreatedNotification notification, CancellationToken ct)

{

Console.WriteLine($"[Log] New car created: {notification.CarId}");

return Task.CompletedTask;

}

}

await \_mediator.Publish(new CarCreatedNotification { CarId = car.Id });

Think of Publish() like **event broadcasting** to multiple subscribers.

## \_mediator.Send() vs \_mediator.Publish()

|  |  |
| --- | --- |
| **\_mediator.Send()** | **\_mediator.Publish()** |
| One-to-One | One-to-Many |
| Waits for a result | No return result (fire-and-forget) |
| Commands and Queries | Events / Notifications |
| IRequest<T> and IRequestHandler<> | INotification and INotificationHandler<> |
| Used in CQRS (Commands/Queries) | Used in Event-driven flows (side effects) |

### Summary Rule

**Need a result?** → Send()  
**Need to trigger effects in other parts of the system?** → Publish()

## **Advanced CQRS + MediatR Summary**

|  |  |
| --- | --- |
| **Topic** | **💬 Description** |
| **CQRS Separation** | Separate **commands** (write) and **queries** (read) into distinct folders/projects |
| **\_mediator.Send()** | For **one-to-one** request/response (e.g., GetCarById, UpdateCar) |
| **\_mediator.Publish()** | For **broadcasting** domain events to multiple handlers (e.g., CarCreated) |
| **<T> in IRequest<T>** | Specifies what the handler returns (Car, string, Unit, etc.) |
| **FluentValidation** | Add request validation with AbstractValidator<T> |
| **Pipeline Behaviors** | Middleware for logging, validation, timing, exception handling |
| **Domain Notifications** | Trigger side effects like emails, logs via INotification handlers |
| **Caching in Queries** | Add IMemoryCache/IDistributedCache to cache frequent query results |
| **Unit Testing** | Test each handler directly with mocks and assertions |
| **Project Structure** | Organize into /Commands, /Queries, /Handlers, /Notifications, /Validators |
| **Microservices Ready** | Each domain can have its own CQRS setup — clean separation |

## WHAT are Events in .NET?

In .NET, **events** are a core part of the **observer pattern**.

### Definition:

An event is a **mechanism for one object (publisher)** to notify other objects (subscribers) that **something has happened**.

They are based on **delegates** and enable **asynchronous or loosely-coupled communication**.

### Think of Events As:

“I want to **let others know** when something happens — they can react however they want.”

## WHY Use Events?

|  |  |
| --- | --- |
| **Use Case** | **Explanation** |
| **Loose coupling** | Publisher and subscriber don't need to know about each other |
| **Broadcasting changes** | Let multiple consumers respond to a single trigger |
| **Real-time reactions** | Automatically update UI, log activity, send notifications, etc. |
| **Scalable design** | Makes code modular, testable, and extendable |

## HOW to Use Events in .NET

.NET supports two major event styles:

### 1. **Traditional .NET Events (Delegate-based)**

#### Step 1: Define Delegate

public delegate void CarCreatedHandler(object sender, CarEventArgs e);

#### Step 2: Declare Event

public class CarService

{

public event CarCreatedHandler CarCreated;

public void CreateCar(Car car)

{

// Save logic

OnCarCreated(car);

}

protected virtual void OnCarCreated(Car car)

{

CarCreated?.Invoke(this, new CarEventArgs(car));

}

}

#### Step 3: Subscribe

carService.CarCreated += (sender, e) =>

{

Console.WriteLine($"Car created: {e.Car.Brand}");

};

#### EventArgs Class

public class CarEventArgs : EventArgs

{

public Car Car { get; }

public CarEventArgs(Car car) => Car = car;

}

### 2. **Event Aggregator Pattern (e.g., using MediatR with INotification)**

This is more **modern and scalable**, especially in ASP.NET Core and CQRS systems.

#### Step 1: Define Event

public class CarCreatedEvent : INotification

{

public string CarId { get; set; }

}

#### Step 2: Raise Event

await \_mediator.Publish(new CarCreatedEvent { CarId = "abc123" });

#### Step 3: Handle Event

public class NotifyAdminHandler : INotificationHandler<CarCreatedEvent>

{

public Task Handle(CarCreatedEvent notification, CancellationToken ct)

{

Console.WriteLine($"[ADMIN] New car created: {notification.CarId}");

return Task.CompletedTask;

}

}

This supports **multiple subscribers** and integrates well with CQRS/MediatR architecture.

## WHEN Should You Use Events?

|  |  |
| --- | --- |
| **Situation** | **Use Event?** |
| Notify other parts of the app when something happens | ✅ Yes |
| Trigger async side effects (logging, emails) | ✅ Yes |
| Tightly connected logic (like one-line calls) | ❌ No |
| Need for multiple reactions to the same action | ✅ Yes |
| UI updates in WinForms/WPF | ✅ Yes |

## Real-World Analogy

Imagine you're in a hotel:

* **Event** = "room service is done"
* **Publisher** = kitchen
* **Subscribers** = front desk, guest, housekeeping

Kitchen calls OnRoomServiceCompleted()  
Everyone listening **reacts independently**.

## Best Practices

|  |  |
| --- | --- |
| **Tip** | **Why?** |
| Use INotification for domain events | Keeps logic modular, decoupled |
| Don't overuse events | Too many events = debugging hell 😅 |
| Avoid logic in UI event handlers | Call services or workflows from handlers |
| Use EventArgs or models | Don’t pass raw data like string or object |
| Use async event handlers (if needed) | Especially for external systems like email/logs |

## Summary

|  |  |
| --- | --- |
| **Concept** | **Description** |
| **Event** | Notification system between objects |
| **Delegate** | Defines the event signature |
| **EventArgs** | Passes data to subscribers |
| **INotification** | Modern .NET Core event model via MediatR |
| **Use cases** | Logging, UI updates, email sending, cascading logic |

## BONUS: Where Events Shine

* Domain-Driven Design (DDD)
* CQRS + MediatR apps
* Microservices (via RabbitMQ, Azure Service Bus)
* WinForms/WPF/UWP/Xamarin/Blazor

**EVENT HANDLER**

an **event** is a **way for a class to provide notifications to other classes or objects when something of interest happens**. It's part of the **Observer Design Pattern**.

Events are usually based on **delegates**.

**Basic Syntax:**

public delegate void MyEventHandler(object sender, EventArgs e);

public class Publisher

{

public event MyEventHandler MyEvent;

public void DoSomething()

{

// Something happens...

OnMyEvent(); // Raise the event

}

protected virtual void OnMyEvent()

{

MyEvent?.Invoke(this, EventArgs.Empty);

}

}

**What is a Custom Event?**

A **custom event** is simply an event that you define **with your own delegate type** or **custom event arguments**.

This is useful when you want to **pass additional information** when the event is raised.

🛠️ **Example with Custom EventArgs:**

public class MyEventArgs : EventArgs

{

public string Message { get; }

public MyEventArgs(string message)

{

Message = message;

}

}

public class Publisher

{

public event EventHandler<MyEventArgs> MyCustomEvent;

public void TriggerEvent()

{

OnMyCustomEvent(new MyEventArgs("Hello from custom event!"));

}

protected virtual void OnMyCustomEvent(MyEventArgs e)

{

MyCustomEvent?.Invoke(this, e);

}

}

public class Subscriber

{

public void Subscribe(Publisher pub)

{

pub.MyCustomEvent += HandleCustomEvent;

}

private void HandleCustomEvent(object sender, MyEventArgs e)

{

Console.WriteLine(e.Message);

}

}

**TL;DR**

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| **Event** | Notification mechanism using delegates |
| **Custom Event** | An event with custom data (custom delegate or EventArgs) |

**Definition of Event Handling :**

Events in ASP.NET raised at the client machine, and handled at the server machine. For example, a user clicks a button displayed in the browser. A Click event is raised. The browser handles this client-side event by posting it to the server.

The server has a subroutine describing what to do when the event is raised; it is called the event-handler. Therefore, when the event message is transmitted to the server, it checks whether the Click event has an associated event handler. If it has, the event handler is executed.

**Event Arguments:**

ASP.NET event handlers generally take two parameters and return void. The first parameter represents the object raising the event and the second parameter is event argument.

# Application and session states:

Application state in ASP.NET is data storage available for all the classes in the application. The state has fast access over the information stored in the database. The state is applicable for all the sessions and users.  
  
**The application state** is not used by multiple servers in the same application. The application data can be used by many threads. The synchronization is must during the access of information from the application state.  
  
There are two events associated with the application state.

1. **Application\_Start:** When user starts application or a website, then the event is raised.

2. **Application\_End:** When user stops application or a website, then the event is raised.

**Session state** helps user store and access values from the user as user navigates the pages in an application. The state checks for the requests from the browser in a limited span of time. The session state is enabled for all the variables.  
The variables are saved in the SessionStateItemCollection object. The Session property is used for exposing the variables.  
There are two events associated with the application state.  
**Session\_Start:** When user requests for the web page from the application, the event is raised.

1. **Session\_End:** It is raised when the session ends.

**Page and control events:**  
  
There are several page and control events in ASP.NET. They are as mentioned below:

1. PreInit: The event is used for recreation of dynamic controls, setting the master page and the theme property, get or set the profile values, verify the IsPostBack property.

2. Init: The event is rasied when all the controls are initialized.

3. InitComplete: The event is raised by the page object.

4. PreLoad: The event is used to perform processing on the page or control before the execution of load event.

5. Load: The event is raised when the control or the page is loaded.

# Event handling using controls:

ASP.NET contains controls which are implemented as classes. They are associated with the events that are executed when an action is performed. There are some built in attributes and event handlers. For responding an event, a event handler is used.

# Default events

Every control in ASP.NET has a default event associated with it. The event handler can be created in the Visual Studio application. User needs to double click on the control present in the design view.  
Some of the controls with their default events are listed below:  
Button Control has Click as the default event

1. BulletedList control has Click as the default event

2. AdRotator control has AdCreated as the default event

3. CheckBox control has the CheckedChanged as the default event

4. Calendar control has the SelectionChanged as the default event

5. CheckBoxList has the SelectedIndexChanged as the default event

6. DataList has the SelectedIndexChanged as the default event.

**Onion Architecture**

**What is Onion Architecture?**

**Onion Architecture** is a layered architectural style that emphasizes **separation of concerns** and **dependency inversion**. It was introduced by **Jeffrey Palermo** to address common issues in traditional architectures like tight coupling and poor testability.

**The Layers of Onion Architecture**

The architecture is shaped like an onion — layers wrapped around a central core:

[4] Infrastructure Layer (Outermost)

[3] Application Services Layer

[2] Domain Services Layer

[1] Domain Entities Layer (Innermost - Core)

**Layer Explanation**

1. **Domain Layer (Core)**
   * Contains: Entities, Value Objects, Interfaces
   * Does *not* depend on anything else
   * This is your system’s **heart** — pure business logic
   * Easy to unit test
2. **Domain Services / Interfaces**
   * Contains: Domain-driven interfaces like IRepository, INotificationService
   * Abstracts out external dependencies
3. **Application Layer**
   * Contains: Use cases, services, DTOs
   * Coordinates the app logic without caring about infrastructure
   * Depends on **domain interfaces**, not implementations
4. **Infrastructure Layer (Outer)**
   * Contains: Data access (e.g. EF Core), APIs, email services, file storage
   * Implements interfaces defined in the core
   * **Depends inward**, NEVER the other way

**Dependency Flow (VERY IMPORTANT):**

**All dependencies must point inward**, toward the Domain Core.  
Never let infrastructure projects reference the domain — this ensures **loose coupling** and **testability**.

**Example: Dependency Chain**

[Infrastructure]

↘ Implements →

[Interfaces]

↘ Used by →

[Application Services]

↘ Operates on →

[Domain Entities]

**Benefits**

* Testable: Domain logic doesn’t rely on databases or frameworks
* Maintainable: Changing infrastructure won’t break your core logic
* Flexible: Easily switch between implementations (e.g., swap SQL Server for MongoDB)
* Clean: Strict boundaries prevent spaghetti code

### ****Dependency Injection Lifetimes in ASP.NET Core****

|  |  |  |
| --- | --- | --- |
| **Lifetime** | **Behavior** | **Use Cases** |
| Transient | A **new instance** is created **every time** it's requested | Lightweight, stateless services like email senders |
| Scoped | A **single instance per HTTP request** | Services handling **business logic**, working with **DbContext** |
| Singleton | A **single instance** for the **entire application lifetime** | Logging, caching, config providers, app-wide settings |

#### Example Registrations

builder.Services.AddTransient<IEmailService, EmailService>();

builder.Services.AddScoped<IUserService, UserService>();

builder.Services.AddSingleton<IConfigService, ConfigService>();

### ****Onion Architecture Overview****

Onion Architecture promotes **clean separation of concerns**, **testability**, and **loose coupling**. All dependencies **point inward** toward the core (Domain layer). The **outer layers** can depend on inner layers, but **not vice versa**.

#### Key Rules:

* Core domain has **no dependencies** on external layers (e.g., web, DB, frameworks)
* Infrastructure and application logic are kept separate
* Easy to **swap** implementation (e.g., change database, UI) without affecting the core

### Onion Layers in Your Project

|  |  |  |
| --- | --- | --- |
| **Layer** | **Contents** | **Responsibility** |
| **1. Application** | Web API controllers, MediatR handlers, DTOs, Repositories | Handles API endpoints, calls to services |
| **2. Domain Services** | Interfaces, service logic (use cases), MediatR handlers | Implements business rules and coordinates actions |
| **3. Domain** | Entities/Models, value objects | Core business logic and rules (pure classes) |
| **4. Shared** | Common models/utilities used across layers | Cross-cutting concerns, shared contracts/models |

### Dependency Flow

[ Application Layer ] → [ Domain Services Layer ] → [ Domain Layer ]

↑ ↑ ↑

Depends inward Depends inward No outward dependency

**4-Layer Onion Architecture Explained**

**1. Application Layer (Outer)**

*Example Project: MyApp.API or MyApp.Web*

|  |  |
| --- | --- |
| **Role** | **Handles** |
| Orchestration layer for user interaction (e.g., API, CLI, etc.) |  |
| Uses controllers, MediatR command/query dispatchers |  |
| Returns DTOs, manages authentication, routing |  |
| Should be thin — avoid any business logic |  |

Depends on: **Domain Services, Shared**

**2. Domain Services Layer**

*Example Project: MyApp.Application*

|  |  |
| --- | --- |
| **Role** | **Coordinates Use Cases (business logic)** |
| Implements application-level logic using domain objects |  |
| MediatR Handlers (if CQRS) live here |  |
| Orchestrates workflows and delegates to domain entities or infrastructure |  |
| Easily testable with mock interfaces |  |

Depends on: **Domain, Shared**

**3. Domain Layer (Core)**

*Example Project: MyApp.Domain*

|  |  |
| --- | --- |
| **Role** | **Contains pure business rules and logic** |
| Core of your application (Entities, ValueObjects) |  |
| Domain interfaces (e.g., IUserRepository) |  |
| Zero dependencies on any framework or tech |  |

Depends on: **Nothing**

**4. Shared Layer (Cross-cutting)**

*Example Project: MyApp.Shared or MyApp.Common*

|  |  |
| --- | --- |
| **Role** | **Reusable models/utilities across all layers** |
| Logging, Error handling, Result types, Base classes |  |
| Validation helpers, contracts, mapping profiles, exceptions |  |

Can be used **by any layer**, including Domain and Infrastructure

**Correct Dependency Flow**

[ Application Layer ] → [ Domain Services Layer ] → [ Domain Layer ]

↘ ↘ ↘

[ Shared Layer ] [ Shared Layer ] [ Shared Layer ]

Only **inward dependencies**  
Shared layer is non-directional and pure utility

Revv-Cars-onion/

Application-Revv.car.repository  
 -Revv.Cars.Web  
Domain - Revv.cars.model

Domain Service-Revv.cars.handler  
shared – Revv.cars.shared

**What is AutoMapper in .NET?**

**AutoMapper** is a **convention-based object-to-object mapper** in .NET. It helps you **automatically map data between objects** — usually between:

* **Entities** (e.g., Car) and
* **DTOs** (Data Transfer Objects like CarDto, CarViewModel)

Instead of writing boilerplate code like this:

var carDto = new CarDto

{

Brand = car.Brand,

Model = car.Model,

Year = car.Year,

...

};

AutoMapper does this automatically:

var carDto = \_mapper.Map<CarDto>(car);

**When to Use AutoMapper?**

Use it when:

* You have a lot of **repetitive mapping code**
* You're working with **DTOs**, especially in APIs
* You want to keep your **domain clean** and isolate presentation logic

**Setup Example**

**1. Install NuGet Package**

dotnet add package AutoMapper

dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection

2. **Create Mapping Profile**

using AutoMapper;

using Revv.cars.Domain.Model;

using Revv.cars.Shared.DTOs;

public class MappingProfile : Profile

{

public MappingProfile()

{

CreateMap<Car, CarDto>(); // Entity to DTO

CreateMap<CarDto, Car>(); // DTO to Entity

}

}

**3. Register AutoMapper in Program.cs**

builder.Services.AddAutoMapper(typeof(Program));

4. **Use in Service or Handler**

public class CarHandler

{

private readonly IMapper \_mapper;

public CarHandler(IMapper mapper)

{

\_mapper = mapper;

}

public CarDto GetDto(Car car)

{

return \_mapper.Map<CarDto>(car);

}

}

**Why It's Great**

✔️ Reduces repetitive code  
✔️ Clean separation between layers  
✔️ Super useful in **Onion Architecture**, especially when transforming between **Domain ↔ Shared ↔ Application**

Submitted By

Anagha N