Entity Framework Core

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ

وَأَنْ لَيْسَ لِلْإِنْسَانِ إِلَّا مَا سَعَىٰ (39) وَأَنَّ سَعْيَهُ سَوْفَ يُرَىٰ (40) ثُمَّ يُجْزَاهُ الْجَزَاءَ الْأَوْفَىٰ (41) وَأَنَّ إِلَىٰ رَبِّكَ الْمُنْتَهَىٰ (42)

📌 My personal accounts links

🔗 LinkedIn	https://www.linkedin.com/in/ahmed-hany-899a9a321?utm_source=share&utm_campaign=share_via&utm_content=profile&utm_medium=android_app
🟢WhatsApp	https://wa.me/qr/7KNUQ7ZI3KO2N1
📘Facebook	https://www.facebook.com/share/1NFM1PfSjc/

📌

What is Deference Between `.Net` & `.Net` Framework & `.Net` Core & `.Net` Framework Core?

👷 .NET

.NET is a software platform made by Microsoft that helps developers build all kinds of applications like:

- Websites 🌐
- Mobile apps 📱
- Desktop programs 🖥️
- Games 🎮
- APIs and microservices ⚙️

It supports multiple programming languages like **C#**, **F#**, and **VB.NET**.

1️⃣ .NET Framework

- This is the **original version**, released in **2002**.
- **Works only on Windows**.
- Used mostly for:
  - Windows desktop apps (like Windows Forms, WPF).
  - Older ASP.NET websites.
- It's still used in many companies for legacy applications.
- But it's **not cross-platform**.

🧠 Example: If you're building a Windows desktop application that runs on Windows 10/11 — that's likely using .NET Framework.

2️⃣ .NET Core

- Introduced in **2016** as a modern version of .NET.
- **Cross-platform** (runs on Windows, Linux, and macOS).
- Faster, lighter, and more flexible.
- Used mostly for:
  - Web applications (ASP.NET Core).
  - APIs.
  - Console applications.
- Supports running **multiple versions side by side** on the same machine.

🧠 Example: A modern website running on a Linux server is likely built with .NET Core.

3️⃣ .NET (5 and above)

- Starting from **.NET 5** (released in 2020), Microsoft decided to merge everything into **one platform**, simply called **".NET"**.
- It's a continuation of .NET Core (no .NET Core 4, they skipped it).
- Supports:

    - Web, desktop, mobile, gaming, IoT, cloud, ...
    - **Cross-platform** (Windows, Linux, macOS).
- This is the **future of .NET**.

🔢 Latest versions:

- .NET 5
- .NET 6 (LTS)
- .NET 7
- .NET 8 (LTS)

🧠 If you're starting a new project — **use .NET 6 or .NET 8** (the latest).

4️⃣ ".NET Framework Core" ❌

This **does NOT exist**.

Some people mix up the names, but there's no official thing called **".NET Framework Core"**.

🧾 Quick Summary Table

Platform	Cross-Platform	Modern?	Latest Version	Use it for
.NET Framework	❌ No	❌ Old	4.8	Windows desktop or legacy projects
.NET Core	✅ Yes	✅ Yes	3.1 (last one)	Modern web & console apps (old core)
.NET (5 and above)	✅ Yes	✅✅ Yes	.NET 8	EVERYTHING! Web, mobile, cloud, etc.
.NET Framework Core	❌ No (fake)	❌ No	❌	❌ Not a real thing

👨‍🏫 My Advice:

- Working on an **old Windows-only project** at a company? You'll probably use **.NET Framework**.
- Starting something **new, modern, fast, and cross-platform**? Use **.NET 6 or .NET 8**.

Release Date	.NET Framework Version	.NET Core and .NET Version
February 2002	.NET Framework 1.0	
April 2003	.NET Framework 1.1	
November 2005	.NET Framework 2.0	
November 2006	.NET Framework 3.0	
November 2007	.NET Framework 3.5	
April 2010	.NET Framework 4.0	
August 2012	.NET Framework 4.5	
October 2013	.NET Framework 4.5.1	
May 2014	.NET Framework 4.5.2	
July 2015	.NET Framework 4.6	
November 2015		.NET Core 1.0 Preview
October 2016	.NET Framework 4.6.2	.NET Core 1.0, 1.1
April 2017	.NET Framework 4.7	.NET Core 2.0 Preview
August 2017		.NET Core 2.0
April 2018	.NET Framework 4.7.2	.NET Core 2.1
September 2018	.NET Framework 4.8 Preview	
December 2018		.NET Core 2.2
April 2019	.NET Framework 4.8	
November 2019		.NET Core 3.0, 3.1
November 2020		.NET 5.0
November 2021		.NET 6.0 (LTS)
November 2022		.NET 7.0
November 2023		.NET 8.0 (LTS)

🗣️ All COMPONENTS OF .NET FRAMEWORK? WHAT ARE THEIR ROLES?

📌 1. Text Editor

is a program where you **write your source code** in a programming language like C#.

Examples:

- Visual Studio
- Visual Studio Code

🔧 What does a Text Editor do?

1. **Code Writing:**
   - It's the place where you type your C# code manually.
   - For example:

```
Console.WriteLine("Hello World");
```

2. **Syntax Highlighting:**
   - It colors different parts of your code for better readability.
     - e.g., keywords like `class`, `if`, `while` appear in different colors.
3. **Auto-completion & IntelliSense:**
   - Helps you write code faster by suggesting variable names, method names, and showing you parameter hints.
   - Example: When you type `Console.Wr`, it may suggest `WriteLine()`.
4. **Error Detection (at typing time):**
   - It can underline syntax errors (like missing semicolons or brackets) before you even compile.
   - Shows red squiggly lines or tooltips for mistakes.
5. **File & Project Management:**
   - Allows you to create and manage multiple files (`.cs`) within a project.
   - Displays a file explorer or project structure.
6. **Integration with other tools:**
   - Many editors support building or running your code directly.
   - You can also use version control like Git inside the editor.

🗂️ What is the output of the Text Editor?

- The main output is your **source code files**, typically with `.cs` extension in C#.
  Example:

```
Program.cs
Doctor.cs
Patient.cs
```

- These files are then passed to the **Compiler**, which is the next component.

📌 🧾 Compiler

✅ What is it?

A **Compiler** is a tool that **translates your human-readable source code** (C# code) into something the computer can understand — specifically, into **Intermediate Language (IL)** code in the .NET world.

For C#, the compiler is typically **Roslyn**, which is built into .NET SDK.

🎯 What does the Compiler do?

1. **Reads your source code**:
   - It takes your `.cs` files (like `Program.cs`, `Doctor.cs`, etc.) as input.
2. **Checks for errors**:
   - If your code has syntax issues or mistakes, the compiler will stop and show errors.
   - For example, forgetting a semicolon `;` or using an undefined variable.
3. **Translates C# to IL (Intermediate Language)**:
   - The compiler turns your readable code into a lower-level, platform-independent language called **CIL (Common Intermediate Language)** or just **IL**.
   - IL is not machine code — it's halfway between C# and actual CPU instructions.
4. **Produces an Assembly**:
   - After compiling, it outputs a file that contains the IL code.
   - This file is usually a `.dll` or `.exe` depending on the project type.

📁 Example:

If you write this C# code:

```
Console.WriteLine("Hello World");
```

The compiler will turn it into IL like this (simplified view):

```
IL_0001: ldstr "Hello World"
IL_0006: call void [System.Console]System.Console::WriteLine(string)
```

Then it writes that into an output file, e.g.:

- `MyProgram.exe`
- or `MyLibrary.dll`

🔍 Summary of Compiler's Responsibilities:

Task	Description
🧠 Understands Code	Reads `.cs` files
⚠️ Checks Code	Validates syntax and logic
🔄 Converts Code	Translates to IL (Intermediate Language)
📦 Outputs File	Produces `.dll` or `.exe` with IL

⚠️ What happens if there's a problem?

If there's any syntax or compile-time error, like:

```
int x = "hello"; // type mismatch
```

The compiler will:

- Stop the build process.
- Show you the error message (like "Cannot implicitly convert type string to int").
- Not produce any output file until you fix it.

📌 Quick Recap:

> Compiler = Code Translator + Code Checker
>
> Input: `.cs` files → **Compiler** → Output: `.dll` or `.exe` with IL code

---

⚙️ CLI (Command-Line Interface)

✅ What is it?

CLI is a **text-based interface** that lets you interact with tools and systems by typing commands.

In the .NET world, the CLI is mainly used for:

- Running the compiler
- Creating projects
- Building, running, testing apps
- Managing dependencies

👉 In .NET, the CLI tool is usually:

```
dotnet
```

💡 Why is CLI important?

The CLI is **how you control the build and execution process** of your .NET application without clicking buttons — just by typing commands.

🧰 What can you do with it?

Command	What it does
`dotnet new`	Creates a new project
`dotnet build`	Compiles your code (calls the compiler internally)
`dotnet run`	Builds AND runs your app
`dotnet publish`	Prepares your app for deployment
`dotnet add package`	Adds NuGet packages (external libraries)

🔁 Relationship with Compiler:

- When you type `dotnet build` or `dotnet run`, the **CLI talks to the Compiler** for you.
- So instead of manually compiling each file, you just run a command like:

  and the CLI:

  ```
  dotnet build
  ```

  - Locates your `.cs` files
  - Sends them to the compiler
  - Stores the output (like `.dll` or `.exe`) in the `bin/` folder

🧪 Example Workflow:

Let's say you created a new console app:

```
dotnet new console -n MyApp
cd MyApp
dotnet run
```

What happens:

1. `dotnet new console` — creates a template project (`Program.cs`, `.csproj`)
2. `dotnet run` — does the following:
   - Compiles the code using Roslyn Compiler
   - Produces `MyApp.dll` (IL code)
   - Passes it to the **CLR** to execute it

📁 Where are the outputs stored?

CLI places build outputs like `.dll` and `.exe` into:

```
/bin/Debug/net7.0/
```

(assuming you're using .NET 7 and Debug mode)

💭 Summary:

> CLI = Your command center 🧠 that controls the compiler and runtime.

- Sends your code to the **Compiler**
- Sends the compiled code to the **CLR**
- Manages everything from project creation to running and publishing

---

📌 🧩 CLR — Common Language Runtime

✅ What is it?

The **CLR** is the **brain of the .NET runtime**. It **takes the compiled IL (Intermediate Language)** code and **executes it on the machine**.

You can think of the CLR as the **.NET virtual machine** — similar to the Java Virtual Machine (JVM) in Java.

🔁 Where does the CLR come in?

Let's say you typed:

```
dotnet run
```

Behind the scenes:

- The CLI builds the code into IL
- The **CLR** takes that IL and **converts it into native machine code** (specific to your CPU)
- Then, it runs that machine code

🧠 What does the CLR do?

Function	Description
🧬 **JIT Compilation**	Converts IL to native machine code (Just-In-Time compilation)
🛡️ **Memory Management**	Automatically allocates and deallocates memory
🚮 **Garbage Collection**	Removes unused objects from memory to free space
🔐 **Security**	Verifies type safety and protects memory access
💥 **Exception Handling**	Manages runtime errors (e.g., division by zero, null reference)
🧪 **Code Verification**	Makes sure the IL is safe and correct before running

🧪 Example of CLR in action

Your IL code says:

```
IL_0001: ldstr "Hello World"
IL_0006: call void [System.Console]System.Console::WriteLine(string)
```

The CLR:

1. Reads that IL
2. Uses **JIT** to turn it into actual machine code instructions
3. The CPU runs that machine code, and you see:

```
Hello World
```

on your console!

🔁 What's JIT (Just-In-Time) Compiler?

- It means the CLR **compiles methods into native code only when needed** (i.e., "just in time").
- So it saves time and memory during execution.

- Output machine code is stored in memory and reused when needed again.

🧪 Fun Features of CLR:

- **Type Safety**: Prevents illegal memory access.
- **Cross-language support**: You can write parts of your app in different .NET languages (e.g., C#, F#, VB) and run them together.
- **Managed Execution**: The CLR controls the full lifecycle of the app.

🧠 Summary:

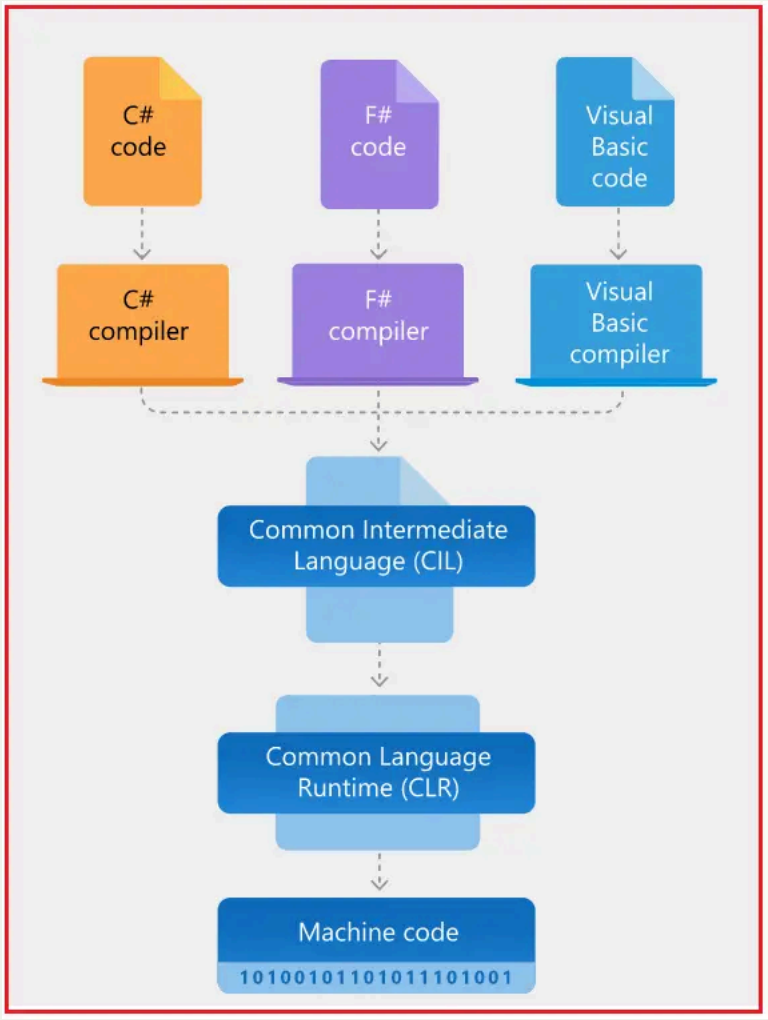
> CLR = The Execution Engine
>
> It runs your program by:

- Turning IL into native code
- Managing memory
- Handling errors
- Ensuring safety

🎯 End-to-End Flow Recap:

Step	Component	What Happens
1️⃣	**Text Editor**	You write `.cs` files
2️⃣	**Compiler**	Translates to IL → creates `.dll` or `.exe`
3️⃣	**CLI**	Runs build & execution commands
4️⃣	**CLR**	Converts IL to native code & runs it 🎉

⚠️


C# code
F# code
Visual Basic code
C# compiler
F# compiler
Visual Basic compiler
Common Intermediate Language (CIL)
Common Language Runtime (CLR)
Machine code
10100101101011101001

📌 What is a technologies Before Entity Framework core ?

the software or program you write. It needs to access a **data source** (like a database or an Excel file).

To do that, the application uses one of the following data access technologies:

- **ODBC**
- **OLE DB**
- **ADO**
- **ADO.NET**

Each of these evolved over time to improve speed, compatibility, and ease of use.

🟦 ODBC (Open Database Connectivity) – 1992

- ODBC is the **oldest** technology
- It provides a **standard API** for accessing **relational databases** (like SQL Server, Oracle, MySQL...).
- It uses SQL to communicate with the database.
- Applications connect to the ODBC Driver → Driver connects to the DB.

➡️ **ODBC → Relational**

Means ODBC is limited to relational databases only.

🟩 OLE DB (Object Linking and Embedding – Database) – 1996

- OLE DB is more **advanced than ODBC**.
- It can access **both relational and non-relational** data sources (like Excel, XML, etc.).
- It uses Microsoft's **COM** technology.
- Requires a "Provider" to connect to a specific data source.

➡️ **OLE DB → Relational + Non-Relational**

It can connect to a wider variety of data sources than ODBC.

🟨 ADO (ActiveX Data Objects) – 1996

- ADO is built **on top of OLE DB** to make it easier for developers.
- Also based on COM (Component Object Model).
- Allows interaction with databases using Recordsets and Commands.

➡️ **ADO → OLE DB**

It uses OLE DB behind the scenes to access the data.

🟦 ADO.NET – 2002

- Part of the **.NET Framework**, and built for use with languages like C# and VB.NET.
- **Does not use COM** (unlike previous ones).
- Uses XML internally and supports **disconnected architecture** (you can work with data without keeping the connection open).

- Works with **CLR** (Common Language Runtime).

➡️ **ADO.NET → CLR + XML**

This shows ADO.NET is a .NET library and uses XML to handle data.

🕒 Evolution Timeline Summary

Year	Technology	Description
1992	ODBC	Standard API for relational databases
1996	OLE DB	Works with both relational and non-relational sources
1996	ADO	Easier COM-based access, built on OLE DB
2002	ADO.NET	Modern .NET-based, supports disconnected model, uses XML

✅ What is ORM?

ORM (Object-Relational Mapping) is a tool or technique that lets you interact with your **database** using **objects and classes** in your programming language instead of writing raw SQL queries.

For example:

- Instead of writing `SELECT * FROM Patients`, you can write something like: `context.Patients.ToList()` (in Entity Framework)

It helps in:

- Saving time
- Reducing errors
- Writing cleaner, more maintainable code

🧩 Types of ORMs (with Examples)

There are different types of ORMs, depending on how **feature-rich** or **lightweight** they are:

🔶 1. Entity Framework Core (EF Core) – ✅ Full ORM (C# / .NET)

- Developed by Microsoft
- Offers full support for:
  - Model creation
  - Migrations
  - LINQ queries
  - Relationships (1-to-many, many-to-many, etc.)
- Best for large and complex applications

Example:

```
var patient = new Patient { Name = "Ali" };
context.Patients.Add(patient);
context.SaveChanges();
```

🔷 2. Dapper – ⚡ Micro ORM (C# / .NET)

- Very lightweight and fast
- Doesn't track changes or generate SQL automatically
- You write your SQL manually, but Dapper maps the result to your class

Example:

```
var patient = connection.Query<Patient>("SELECT * FROM Patients WHERE Id = @Id", new { Id = 1 });
```

- Best when you want speed and have control over SQL

🔶 3. NHibernate – 🧠 Full ORM (C# / .NET)

- Older than EF Core, very powerful
- Allows advanced mappings and configurations
- Supports lazy loading, caching, inheritance, ...

- More configuration-heavy than EF Core

Example:

```
session.Save(new Patient { Name = "Omar" });
```

- Suitable for enterprise-level apps needing advanced ORM features

---

🆚 Summary Comparison

Feature	EF Core	Dapper	NHibernate
Type	Full ORM	Micro ORM	Full ORM
Tracking	Yes	No	Yes
Performance	Good	Excellent	Good
Ease of use	Easy	Easy (for SQL users)	Moderate
Configurations	Convention-based	Manual SQL	XML or Fluent
Best for	General .NET apps	High-performance apps	Advanced scenarios

---

If you're building something like your **hospital management system**, I recommend:

- ✅ **EF Core** if you want full features and work mostly with C# and LINQ.
- ⚡ **Dapper** if performance is critical and you prefer writing SQL yourself.
- 🧠 **NHibernate** if you need complex mappings and enterprise-level flexibility.

📌

⚡ Why is ADO.NET faster than ORM?

ADO.NET is generally faster than an **ORM** like **EF Core**, **NHibernate**, or **Dapper** (to a smaller extent) because it works **closer to the metal**, meaning it:

✅ 1. Avoids Abstraction Overhead

ORMs add a layer of abstraction to help developers work with data as objects. This abstraction:

- Maps database rows to classes
- Tracks changes
- Builds and optimizes SQL queries behind the scenes

> ADO.NET, on the other hand, sends SQL directly to the database and reads results without conversion or tracking.

✅ 2. No Change Tracking

ORMs like EF Core automatically **track object changes** (so they can generate update queries automatically), which consumes **CPU and memory**.

> ADO.NET doesn't track anything — you control all SQL manually.

✅ 3. Direct SQL Execution

In ADO.NET:

- You write raw SQL and execute it
- You manage the connection, command, and data reading yourself

This means:

- Less CPU work
- Less memory usage
- No LINQ-to-SQL conversion
- No automatic model binding

✅ 4. Fewer Layers

ADO.NET uses fewer internal classes and logic, making execution faster. ORMs go through:

- Entity tracking
- Expression trees
- LINQ translation
- Caching mechanisms
- Mapping layers

All of which add overhead.

🧪 Real-Life Analogy

Imagine:

- **ADO.NET** is like writing a message by hand and handing it directly to someone.
- **ORM** is like telling an assistant, who formats the message, translates it, adds context, and then sends it for you.

The second way is easier — but slower!

📌 When to Use ADO.NET vs ORM

Use ADO.NET When	Use ORM When
You need **maximum performance**	You want **productivity and cleaner code**
You're writing **simple queries**	You're building a **large, maintainable app**
You want full control over SQL	You want to avoid manual SQL
You're doing **batch processing or bulk inserts**	You want object-oriented access to data

✅ Summary

Feature	ADO.NET	ORM (e.g., EF Core)
Speed	⚡ Faster	Slower (due to overhead)
Ease of use	❌ Manual	✅ Easy to use
Code readability	❌ Low	✅ High
Abstraction	❌ None	✅ High-level
Change tracking	❌ No	✅ Yes

Let's Starting in Entity Framework Core

✅ What is Entity Framework Core?

Entity Framework Core (EF Core) is a **modern, open-source, object-relational mapper (ORM)** for **.NET** (developed by Microsoft) and it is lightweight

It allows you to:

- Interact with your **relational database** using **C# classes and LINQ** instead of raw SQL.
- Perform **CRUD operations** (Create, Read, Update, Delete) using objects.
- Automatically handle relationships, migrations, and schema generation.

> It's the recommended ORM for ASP.NET Core applications.

🔧 What EF Core Does Behind the Scenes

- Maps your C# classes to database tables
- Converts LINQ queries to SQL
- Tracks changes to objects and saves them to the database
- Handles relationships like one-to-many, many-to-many, etc.
- Supports migrations to keep your database schema in sync with your models

What is a Deference between `.Net` and Entity Framework?

`.Net` : it is a platform for development APPS

Entity Framework /Core: For Data Access (ORM)

🆚 EF vs EF Core

Feature	**Entity Framework (EF)**	**Entity Framework Core (EF Core)**
Platform	Windows only	Cross-platform (Windows, Linux, macOS)
Performance	Slower	Faster and lightweight
Database Support	Mostly SQL Server	Many databases (SQL Server, SQLite, MySQL, PostgreSQL...)
Design Approach	EDMX (Designer-based) + Code-First	Code-First only
Mapping System	Uses EDM with XML mapping	Uses Fluent API and Data Annotations
Migration Support	Limited	Built-in powerful migrations
Change Tracking	Yes	Yes, more flexible and optimized
Modularity	Monolithic	Modular (can add only what you need)
LINQ Support	Yes	Yes (more modern and efficient)

In short:

- EF Core is newer, faster, and more flexible.
- EF (classic) is older and works well with older .NET Framework projects.

🌟 Entity Framework Core Model

When using EF Core, you work with **two main parts**:

🧱 1. Entities

- These are **your C# classes**.
- Each class represents a **table** in your database.
- Example: A `Student` class becomes a `Students` table.

📦 2. Context Object (`DbContext`)

- This is like the **bridge(Session)** between your code and the database.
- It opens a **session** with the database.
- the session is between Entity and Table
- You use it to **query**, **add**, **update**, or **delete** data.

✅ What is an Entity in Entity Framework Core?

An **Entity** is just a **C# class** that represents a **table** in your **database**.

Each **property** in the class becomes a **column** in the table.

🧾 Example of a Table in a Database:

Let's say you have a table in the database called:

`Patients` Table

Id	Name	Age	Email
1	Omar said	30	omar@example.com
2	Ahmed Ashraf	20	ahmed@example.com

🔄 Entity Mapping in C#:

This is how you represent (map) that table using a C# class (Entity):

```
public class Patient
{
    public int Id { get; set; }         // Maps to Id column
    public string Name { get; set; }    // Maps to Name column
    public int Age { get; set; }        // Maps to Age column
    public string Email { get; set; }   // Maps to Email column
}
```

> This Patient class is an Entity.

🎯 Summary:

- **Table name**: `Patients`
- **Entity class**: `Patient`
- Each row in the table = one object of `Patient`
- Each column = one property in the class

✅ What is `DbContext`?

`DbContext` is a class that acts like a **bridge** (or a session) between your **C# code** and your **database**.

It:

- Manages connection to the DB
- Lets you **query**, **add**, **update**, and **delete** data
- Tracks changes to entities

🧩 How to use `Patient` with `DbContext`

1️⃣ Step 1: Create a `Patient` entity

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

2️⃣ Step 2: Create a custom `DbContext` class

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    // This tells EF Core: “There’s a table in the database called Patients”
    public DbSet<Patient> Patients { get; set; }

    // Optional: Configure the DB provider and connection string
    protected override void OnConfiguring
    (DbContextOptionsBuilder optionsBuilder)
    {
        // Replace with your own DB connection string
        optionsBuilder.UseSqlServer("Server=.;Database=MyHospitalDB;Trusted_Connection=True;");
    }
}
```

3️⃣ Step 3: Use it in your app

```
using (var context = new ApplicationDbContext())
{
    // Add a patient
    var newPatient = new Patient
    {
        Name = "Ali",
        Age = 40,
        Email = "ali@example.com"
    };
```

```
    context.Patients.Add(newPatient);  // Mark for insert
    context.SaveChanges();             // Executes insert to DB

    // Query patients
    var patients = context.Patients.ToList();
}
```

🔁 Summary:

- `DbContext` = gateway to the database
- `DbSet<Patient>` = represents the table `Patients`
- You can now use C# to work with the database easily — without writing SQL.

What is a connection string?

A connection string is a string that contains information needed to establish a connection to a database or another data source. It typically includes details such as the type of data source, the server address, the database name, authentication credentials (like username and password), and other parameters that are necessary for the connection.

Here's a breakdown of common components in a connection string:

1. **Data Source/Server**: The name or network address of the database server.
2. **Initial Catalog/Database**: The name of the database to connect to.
3. **User ID/Username**: The username for authentication.
4. **Password**: The password for authentication.
5. **Integrated Security**: Indicates whether Windows Authentication is used (common in SQL Server).
6. **Provider**: Specifies the database provider (e.g., SQL Server, MySQL, Oracle).
7. **Additional Parameters**: Other options like connection timeouts, encryption settings, etc.

Example Connection Strings

1. **SQL Server:**

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

2. **MySQL:**

```
Server=myServerAddress;Database=myDataBase;Uid=myUsername;Pwd=myPassword;
```

3. **PostgreSQL:**

```
Host=myServerAddress;Database=myDataBase;Username=myUsername;Password=myPassword;
```

4. **SQLite:**

```
Data Source=myDatabaseFile.db;Version=3;
```

5. **Oracle:**

```
Data Source=myServerAddress/myDataBase;User Id=myUsername;Password=myPassword;
```

Usage

Connection strings are used in various programming environments and frameworks to connect applications to databases. For example, in .NET applications, connection strings are often stored in configuration files (like `app.config` or `web.config`) and accessed via the `ConfigurationManager` class.

Security Considerations

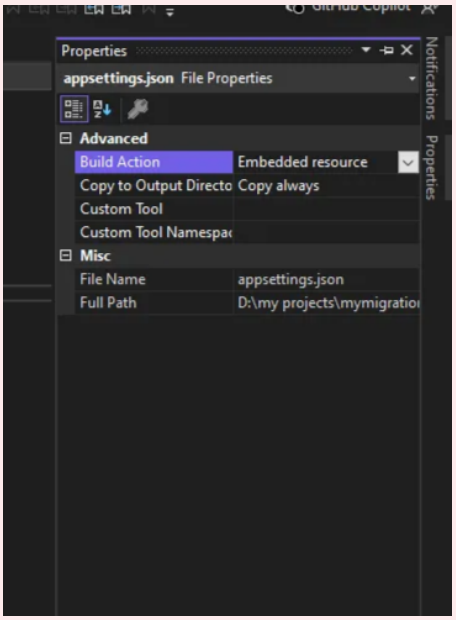
- **Sensitive Information**: Connection strings often contain sensitive information like usernames and passwords. It's important to secure them, for example, by using encrypted configuration files or environment variables.
- **Avoid Hardcoding**: Avoid hardcoding connection strings directly in your source code. Instead, use configuration files or secure vaults.

🔑 How can create Configuration in your Project with `EFCore` ?

1. add `Json` configuration file to store your connection string

```
{
  "constr": "Server =  server name ; Database = DB Name ; Integrated Security = SSPI ; TrustServerCertificate = True"
}
```

2. install Nugget Package `Microsoft.Extensions.Configuration`
3. install Nugget Package `Microsoft.Extensions.Configuration.Json`
4. add setting to your `json` configuration file to embedded it with Metadata



5. Create an Entity

```
namespace EFCore1
{
    public class Patient
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public string Email { get; set; }

        public override string ToString()
        {
            return $"{Id} : {Name}  ⇒ age : {Age}  with Email : {Email}";
        }
    }
```

```
}
```

6. Install Packages for SQL Provider : `Microsoft.EntityFrameworkCore.SqlServer`
7. add your `DBcontext` with internal configuration (parameter less Constructor)

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EFCore1
{
   internal class AppDBContext:DbContext
   {

      public DbSet<Patient> Patients { get; set; }
      //here you tell DBcontext of the Entity that can create a session to it

      protected override void OnConfiguring
      (DbContextOptionsBuilder optionsBuilder)
      {
         base.OnConfiguring(optionsBuilder);

         var configuration = new ConfigurationBuilder()
             .AddJsonFile("appsettings.json")
               .Build();

         var constr = configuration.GetSection("constr").Value;

         optionsBuilder.UseSqlServer(constr);
      }
   }
}
```

What is a best practice for using `DBContext`?

`DBContext` is apply Disposable then use it in `useing` blocks

```
using()
{
//code here
}
```

Let’s apply simple code for retrieve data from data base :

```
using Microsoft.Extensions.Configuration;

namespace EFCore1
{
   internal class Program
   {
      static void Main(string[] args)
      {

         using(AppDBContext context=new AppDBContext()) //apply IEnummerable then can apply foreach
         {

            foreach (var patient in context.Patients)
            {
               Console.WriteLine(patient);
            }

            Console.WriteLine
            ($"patient 3 ⇒ {context.Patients.FirstOrDefault(x⇒x.Id==3)}");
         }

         Console.ReadKey();
      }
   }
}
```

> Output:
>
> 1 : ali ⇒ age : 21 with Email : ali@gmail.com
> 2 : omar ⇒ age : 22 with Email : omar@gmail.com
> 3 : yasser ⇒ age : 22 with Email : yasser@gmail.com
> 4 : kamel ⇒ age : 30 with Email : kamel@gmail.com
> 5 : ahmed ⇒ age : 20 with Email : ahmed@gmail.com
> patient 3 ⇒ 3 : yasser ⇒ age : 22 with Email : yasser@gmail.com

💡 Let’s Inserted Data :

.

```
using Microsoft.Extensions.Configuration;

namespace EFCore1
{
   internal class Program
   {
      static void Main(string[] args)
      {

         using(AppDBContext context=new AppDBContext()) //apply IEnummerable then can apply foreach
         {
            context.Patients.Add(new Patient
            {
               Id = 6,
               Name = "ola",
               Age = 26,
               Email = "ola@gmail.com"
            });

            context.SaveChanges();

         }


         using(AppDBContext context=new AppDBContext())
         {
            Console.WriteLine($" added patient 6 : {context.Patients.FirstOrDefault(x ⇒ x.Id == 6)}");
         }


         Console.ReadKey();
      }
   }
}
```

> added patient 6 : 6 : ola ⇒ age : 26 with Email : ola@gmail.com

💡 Let’s Update Data :

```
using Microsoft.Extensions.Configuration;

namespace EFCore1
{
   internal class Program
   {
      static void Main(string[] args)
      {

         using(AppDBContext context=new AppDBContext()) //apply IEnummerable then can apply foreach
         {
           var patient6=context.Patients.FirstOrDefault(x⇒x.Id==6);


            patient6.Age = 30;

            context.SaveChanges();

         }


         using(AppDBContext context=new AppDBContext())
         {
            Console.WriteLine($" after update patient 6 : {context.Patients.FirstOrDefault(x⇒x.Id==6)}");
         }


         Console.ReadKey();
      }
   }
}
```

> after update patient 6 : 6 : ola ⇒ age : 30 with Email : ola@gmail.com

Let’s Delete Data :

```
using Microsoft.Extensions.Configuration;

namespace EFCore1
{
   internal class Program
   {
      static void Main(string[] args)
      {

         using(AppDBContext context=
         new AppDBContext()) //apply IEnummerable then can apply foreach
         {
           var patient6=context.Patients.FirstOrDefault(x⇒x.Id==6);


           context.Patients.Remove(patient6);

            context.SaveChanges();


            foreach (var item in context.Patients)
            {
               Console.WriteLine(item);
            }

         }
         Console.ReadKey();
      }
   }
}
```

> 1 : ali ⇒ age : 21 with Email : ali@gmail.com
> 2 : omar ⇒ age : 22 with Email : omar@gmail.com
> 3 : yasser ⇒ age : 22 with Email : yasser@gmail.com
> 4 : kamel ⇒ age : 30 with Email : kamel@gmail.com
> 5 : ahmed ⇒ age : 20 with Email : ahmed@gmail.com

💡 Let’s apply Transaction

```
using (AppDBContext context = new AppDBContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            var fromwallet = context.Patients.FirstOrDefault(x => x.Id == 1);
            var towallet = context.Patients.FirstOrDefault(x => x.Id == 2);
            var amount = 500m;

            if (fromwallet == null || towallet == null)
            {
                Console.WriteLine("One of the patients not found.");
                return;
            }

            if (fromwallet.wallet < amount)
            {
                Console.WriteLine("Insufficient funds.");
                return;
            }

            fromwallet.wallet -= amount;
            towallet.wallet += amount;

            context.SaveChanges();
            transaction.Commit();
        }
        catch (Exception ex)
        {
            transaction.Rollback();
            Console.WriteLine($"Error occurred: {ex.Message}");
        }
    }
}
```


SELECT TOP (1000) [Id]
,[Name]
,[Age]
,[Email]
,[wallet]
FROM [EFDB].[dbo].[Patients]
131 %
Results
Messages
Id Name Age Email wallet
1 1 ali 21 ali@gmail.com 7500
2 2 omar 22 omar@gmail.com 8500
3 3 yasser 22 yasser@gmail.com 8000
4 4 kamel 30 kamel@gmail.com 8000
5 5 ahmed 20 ahmed@gmail.com 8000

what is a types of Configurations that can apply ?

1. internal configurations ⇒ we applied in last Example
2. External Configurations : by use parameter constructor of `Dbcontext`

that used in `ASP.Net`

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EFCore1
{
   internal class AppDBContext:DbContext
   {

      public DbSet<Patient> Patients { get; set; }

      public AppDBContext(DbContextOptions contextOptions):base(contextOptions) { }

   }
}
```

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System.Transactions;

namespace EFCore1
{
   internal class Program
   {
      static void Main(string[] args)
      {
         //External Configurations

         var config=new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();
         var connectionstring = config.GetSection("constr").Value;

         var dbcontextoptionbuilder = new DbContextOptionsBuilder();
         dbcontextoptionbuilder.UseSqlServer(connectionstring);


         var options= dbcontextoptionbuilder.Options;
```

```
            using(AppDBContext context=new AppDBContext(options))
            {
               foreach (var p in context.Patients)
               {
                  Console.WriteLine(p);
               }

            }

            Console.ReadKey();
         }
      }
   }
```

3. by Dependence Injection

> You use Dependency Injection to make your code easier to manage, test, and change.
>
> It lets the system give your class what it needs, instead of your class creating it by itself.

✅ Makes testing easier

✅ Keeps code clean

✅ Helps your app grow without getting messy

```
namespace EFCore1
{
   internal class AppDBContext:DbContext
   {

      public DbSet<Patient> Patients { get; set; }

      public AppDBContext(DbContextOptions contextOptions):base(contextOptions) { }

   }
}
```

```
      static void Main(string[] args)
      {
         //External Configurations

         var config=new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();
         var connectionstring = config.GetSection("constr").Value;


         var service = new ServiceCollection();
         service.AddDbContext<AppDBContext>(options⇒
```

```
        options.UseSqlServer(connectionstring));

        IServiceProvider serviceProvider=service.BuildServiceProvider();

        using(var context=serviceProvider.GetService<AppDBContext>())
        {
         foreach (var p in context.Patients)
         {
            Console.WriteLine(p);
         }
       }

        Console.ReadKey();
      }
```

4. **Context Factory**

```
namespace EFCore1
{
   internal class AppDBContext:DbContext
   {

      public DbSet<Patient> Patients { get; set; }

      public AppDBContext(DbContextOptions contextOptions):base(contextOptions) { }

   }
}
```

```
      static void Main(string[] args)
      {
         //External Configurations

         var config=new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();
         var connectionstring = config.GetSection("constr").Value;


         var service = new ServiceCollection();
         service.AddDbContextFactory<AppDBContext>(options⇒
         options.UseSqlServer(connectionstring));

         IServiceProvider serviceProvider=service.BuildServiceProvider();


         var conrextfactory=serviceProvider.
            GetService<IDbContextFactory<AppDBContext>>();

         using(var context=conrextfactory.CreateDbContext())
```

```
        {
            foreach (var p in context.Patients)
            {
                Console.WriteLine(p);
            }
        }

        Console.ReadKey();
    }
```

but this Context Factory here can not help us if

- you create simple Console App
- Trying `Dbcontext` manual outside ASP.NET

Best Practices for use Context factory :

- It solves the problem when the `DbContext` is in a separate project from the main application like you Class Library .
- No need to run the actual application or modify `Program.cs` .
- It gives you full control over how the `DbContext` is created.
- it working in Design
- help you in Migration

add new class in Your Class Library project

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.Configuration;
using System.IO;

public class ApplicationDbContextFactory :
IDesignTimeDbContextFactory<ApplicationDbContext>
{
    public ApplicationDbContext CreateDbContext(string[] args)
    {
        // Load appsettings.json manually
        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            // change dependance on run location
            .AddJsonFile("appsettings.json")
            .Build();

        var optionsBuilder =
        new DbContextOptionsBuilder<ApplicationDbContext>();
        optionsBuilder.UseSqlServer(configuration
        .GetConnectionString("DefaultConnection"));

        return new ApplicationDbContext(optionsBuilder.Options);
    }
}
```

🔍 Full Comparison Table

Method	When to Use	Works at Runtime?	Works at Design Time (Migrations)?	Easy to Test?	Complexity
1. Internal Configuration	Define connection and config inside the `DbContext` class itself	✅ Yes	❌ No	❌ No	✅ Simple
2. External Configuration	Configure in `Program.cs` using `AddDbContext`	✅ Yes	❌ No	✅ Yes	✅ Simple
3. Dependency Injection (DI)	Use the framework to inject `DbContext` into your classes	✅ Yes	❌ No	✅ Yes	✅ Simple
4. Context Factory	Use when `DbContext` is in a separate class library (for design-time support)	❌ No	✅ Yes	✅ Yes	🔄 Medium

✅ What is a Best practices for adding configurations in EF Core?

1. **Use** `IDesignTimeDbContextFactory`
   ➤ For Migrations in class library projects.
   ➤ Helps EF Core create `DbContext` at design-time (e.g., `dotnet ef migrations add`).
2. **Use** `OnConfiguring` **only for simple/testing cases**
   ➤ Avoid it in real apps — it mixes configuration inside `DbContext`.
3. **Use Dependency Injection (DI)** in your main project
   ➤ Register `DbContext` using `AddDbContext` in `Program.cs`.
   ➤ Read connection string from `appsettings.json`.
4. **Use External Configuration classes (Fluent API)**
   ➤ Create separate `EntityTypeConfiguration` classes for each entity.
   ➤ Apply them using `modelBuilder.ApplyConfigurationsFromAssembly(...)`.

🟢 **Recommended combo in real projects:**

- `DbContext` + external configuration classes
- `IDesignTimeDbContextFactory` for migrations
- `AddDbContext` with DI for runtime configuration

✅
Best Practice (with API + Class Library): Use External Configurations + Context Factory

✅ Question:

If you have your entities and `DbContext` inside a Class Library, and you only used External Configurations, can you run a migration? Why or why not?

🟢 Answer:

❌ No, you can't run a migration.

Because External Configurations only configure the entities, but they don't tell EF Core how to create the `DbContext` at migration time.

You need to use either `OnConfiguring` or a `Context Factory` so EF Core knows how to build the `DbContext` when running migrations.

✅ What is the `DbContext` Life Cycle?

- **Create**: `DbContext` is created
- **Track**: EF watches entities you query/change
- `SaveChanges`: Changes go to the database
- **Dispose**: `DbContext` is removed from memory

💡 What is `DbContextPool` in EF Core?

`DbContextPool` is a **performance feature** that **reuses DbContext instances** instead of creating a new one every time.

✅ How It Works:

- Normally, when you use `AddDbContext`, EF Core creates a **new** `DbContext` for each request.
- With `AddDbContextPool`, EF Core keeps a **pool (cache)** of pre-created `DbContext` instances.
- When a request comes in, EF Core **reuses** an instance from the pool (if available), instead of making a new one.

⚡ Benefits:

Feature	Explanation
✅ **Faster**	Reduces the cost of creating and disposing many DbContext instances
✅ **Efficient**	Better memory and resource usage
✅ **Good for high-traffic apps**	Especially useful in APIs or web apps with lots of requests

🛑 Notes:

- The `DbContext` must be **clean** when returned to the pool (EF Core resets it).
- You **should not store** the `DbContext` in static fields or use it across multiple threads.
- Flush your cache of `DbContexts` after full and recreate new dbcontext

```
static void Main(string[] args)
{
   //External Configurations

   var config=new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();
   var connectionstring = config.GetSection("constr").Value;


   var service = new ServiceCollection();
   service.AddDbContextPool<AppDBContext>(options⇒
   options.UseSqlServer(connectionstring));

   IServiceProvider serviceProvider=service.BuildServiceProvider();




   using(var context=serviceProvider.GetService<AppDBContext>())
   {
      foreach (var p in context.Patients)
      {
         Console.WriteLine(p);
      }
   }
```

```
        Console.ReadKey();
    }
```

💡 💼 **Interview Question:**

Q: Can you explain the different approaches for creating a database using Entity Framework?

✅ Answer:

Yes, there are three main approaches for creating a database using Entity Framework:

1. **Code First:**

   In this approach, you write your C# model classes first. Entity Framework uses those classes to generate the database schema. It's commonly used for new applications where the database structure evolves with the code. Migrations are used to manage changes over time.

2. **Database First:**

   Here, the process starts with an existing database. You generate C# entity classes from the database using tools like `Scaffold-DbContext` (Reverse Engineering) in EF Core. This approach is ideal for legacy systems or when the database is managed separately by a DBA.

3. **Model First:**

   This approach involves designing the database visually using an Entity Framework Designer. From the visual model, both the database and C# classes are generated. This is mostly supported in older versions of EF and is not recommended in EF Core.

💡 💼 **Interview Question:**

Q: What are the two main approaches to configure entity models in Entity Framework Core, and when would you use each?

✅ Answer:

Entity Framework Core provides two main approaches for configuring entity models:

1. Conventions

- These are **default rules** that EF Core applies automatically based on the class and property names.
- For example, EF will automatically create a primary key if it finds a property named `Id` or `ClassNameId`.
- Conventions are great for quick development and when your models follow standard naming patterns.

✅ Entity Framework Core – Default Conventions

1. Primary Key Convention

If a property is named `Id` or `{EntityName}Id`, EF Core automatically sets it as the primary key.

2. Foreign Key Convention

If there is a navigation property and a corresponding `{NavigationPropertyName}Id`, EF assumes it's a foreign key.

3. Relationship Convention

EF Core infers relationships (one-to-one, one-to-many) based on navigation properties and foreign key presence.

4. Cascade Delete Convention

For required relationships, EF Core enables cascade delete by default.

For optional relationships, it sets delete behavior to `ClientSetNull`.

5. Naming Convention

- Table names match class names.
- Column names match property names.

6. Data Type Convention

EF Core automatically maps .NET property types to suitable SQL data types.

7. Nullability Convention

- Non-nullable types are treated as **required**.
- Nullable types are treated as **optional**.

8. Index Convention

EF Core automatically creates indexes on foreign key properties.

9. Shadow Property Convention

If a foreign key exists but isn't explicitly defined, EF creates an internal hidden (shadow) property for it.

10. Pluralization

EF Core does **not pluralize table names by default** — table names stay the same as entity class names unless overridden.

2. Configuration

- This involves explicitly configuring models using data annotations or the `ModelBuilder` inside `OnModelCreating` or using **separate configuration classes** via `IEntityTypeConfiguration<T>`.
- It gives you **fine-grained control** over table names, keys, relationships, constraints, and more.
- Recommended when you need to override default conventions or handle more complex scenarios.

✅ Example Use:

- Use **conventions** for simple models to save time.
- Use **configurations** when you need to define things like composite keys, table names, or relationships explicitly.

💡 What are the Types for add Configurations that can apply ?

✅ 1. Data Annotations

✔️ Definition:

Data Annotations are attributes applied directly to entity classes and their properties to configure how EF Core maps them to the database.

✔️ Advantages:

- Simple and quick for basic configurations.
- Easy to understand and use.
- Keeps configuration close to the model.

❌ Disadvantages:

- Limited to simple configurations.
- Not suitable for complex relationships.
- Can clutter the entity classes.

🧪 Example:

```
public class Student
{
    [Key]
    public int StudentId { get; set; }

    [Required]
    [MaxLength(100)]
    public string FullName { get; set; }

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

    [ForeignKey("Department")]
    public int DepartmentId { get; set; }

    public Department Department { get; set; }
}
```

✅ 2. Fluent API

✔️ Definition:

Fluent API is used in `OnModelCreating` method or via `EntityTypeConfiguration` classes to configure models using code rather than attributes.

✔️ Advantages:

- Great for advanced configurations.

- Separates configuration from the model (cleaner code).
- More powerful and flexible.

❌ Disadvantages:

- Slightly more verbose than data annotations.
- Requires more setup.

🧪 Example:

```
public class Student
{
    public int StudentId { get; set; }
    public string FullName { get; set; }
    public int Age { get; set; }
    public int DepartmentId { get; set; }
    public Department Department { get; set; }
}
```

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>(entity =>
        {
            entity.HasKey(s => s.StudentId);
            entity.Property(s => s.FullName)
                .IsRequired()
                .HasMaxLength(100);
            entity.Property(s => s.Age)
                .HasDefaultValue(18);
            entity.HasOne(s => s.Department)
                .WithMany()
                .HasForeignKey(s => s.DepartmentId)
                .OnDelete(DeleteBehavior.Cascade);
        });
    }
}
```

✅ 3. Group Configuration (EntityTypeConfiguration Class)

✔️ Definition:

Instead of writing configurations in `OnModelCreating`, you can group them into separate classes using the `IEntityTypeConfiguration<T>` interface.

✔️ Advantages:

- Keeps `DbContext` clean.
- Organizes configurations per entity.
- Useful in large projects.

❌ Disadvantages:

- Requires more files (per entity).
- Slightly more setup work.

🧪 Example:

```
public class Student
{
   public int StudentId { get; set; }
   public string FullName { get; set; }
   public int Age { get; set; }
   public int DepartmentId { get; set; }
   public Department Department { get; set; }
}
```

✅ StudentConfiguration.cs

```
public class StudentConfiguration : IEntityTypeConfiguration<Student>
{
   public void Configure(EntityTypeBuilder<Student> builder)
   {
      builder.HasKey(s => s.StudentId);
      builder.Property(s => s.FullName)
            .IsRequired()
            .HasMaxLength(100);
      builder.Property(s => s.Age)
            .HasDefaultValue(18);
      builder.HasOne(s => s.Department)
            .WithMany()
            .HasForeignKey(s => s.DepartmentId)
            .OnDelete(DeleteBehavior.Cascade);
   }
}
```

✅ Inside `DbContext` :

```
//for one specific congig class
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   modelBuilder.ApplyConfiguration(new StudentConfiguration());
}

//for all config classes in same assemply

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   base.OnModelCreating(modelBuilder);
   modelBuilder.ApplyConfigurationsFromAssembly
   (typeof(StudentConfiguration).Assembly);
}
```

🔁 Comparison Summary Table:

Feature	Data Annotations	Fluent API	Group Configuration
Location	Inside the model	In `OnModelCreating`	Separate class per entity
Flexibility	❌ Limited	✅ Very flexible	✅ Very flexible
Clean Code	❌ Can clutter model	✅ Better separation	✅ Best organization
Suitable for large apps?	❌ No	✅ Yes	✅ Yes
Complex relationships	❌ Limited	✅ Full support	✅ Full support

What is a Reverse Engineering in Entity Framework Core (EF Core)

1. Definition

Reverse Engineering in **EF Core** is the process of generating **C# entity classes and a** `DbContext` from an existing database schema. This is commonly used in the **Database-First** approach, where a project starts with an existing database instead of defining the models first in code.

2. How It Works

The process involves using **EF Core tools** to scaffold models from the database schema:

1. **Install EF Core Tools** :

`Install-Package Microsoft.EntityFrameworkCore.Tools`

2. **Install EF Core Design:**

`Install-Package Microsoft.EntityFrameworkCore.Design`

3. **Install SQL Provider :**

`Install-Package Microsoft.EntityFrameworkCore.SqlServer`

4. **Run the Reverse Engineering Command:**

```
scaffold-DbContext "Your_Connection_String"  Your Provider
```

5. **Generated Code:**
   - A `DbContext` class that maps database tables to entity classes.
   - Entity classes representing each table in the database.

3. Limitations

- **Complex Relationships:** Some database relationships (like complex foreign keys) may not be generated accurately.
- **Limited Customization:** The generated models may require manual modifications to align with best coding practices.
- **Scaffolded Code Overwrite:** If you rerun the scaffold command, it may overwrite existing changes unless handled carefully.
- **Stored Procedures and Views:** EF Core does not natively scaffold **stored procedures** and **views** without additional configuration.
- **Performance Considerations:** Large databases may lead to excessive code generation, making maintenance difficult.

How to Improve Reverse Engineering Results?

- Use `-DataAnnotations` to apply attributes instead of Fluent API.
- Use `-Context <ContextName>` to specify a custom context name.
- To specify schema use ⇒ -Schema `schemaName`
- To specify tables use ⇒ -Tables `TableName`
- To add your Context in Folder and your Entities in Dir use - `ContextDir` Folder Name - `OutputDir` Folder of Entities
- Manually refine the generated models to improve structure and maintainability.

How can apply mapping with `EFCore` ?

Let’s apply a Example to understand , this Example will mapping

1. Three Entities has deference Relations
2. one Ignored Entity that not mapped in Database only Exists in Memory
3. include Entity with eggier loading not with `Dbset`

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyHospitalEFCore
{
    public class MyDbContext : DbContext
    {
        public DbSet<Doctor> Doctors { get; set; }
        public DbSet<Patient> Patients { get; set; }
        public DbSet<Appointment> Appointments { get; set; }

        // Ignored Entity (Will not be mapped to DB)
        public DbSet<TemporaryData> TemporaryDatas { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Your_Connection_String");
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // Configure relationships
            modelBuilder.ApplyConfiguration(new DoctorConfiguration());
            modelBuilder.ApplyConfiguration(new PatientConfiguration());
            modelBuilder.ApplyConfiguration(new AppointmentConfiguration());

            // Ignore an entity
            modelBuilder.Ignore<TemporaryData>();

            // Define a View
            modelBuilder.Entity<DoctorView>().ToView("DoctorView").HasNoKey();

            // Table-Valued Function Mapping
            modelBuilder.Entity<AppointmentSummary>().HasNoKey().ToFunction("GetAppointmentSummary");
        }
    }

    // Entities
    public class Doctor
    {
        public int Id { get; set; }
```

```
    public string Name { get; set; }
    public virtual ICollection<Patient> Patients { get; set; } = new List<Patient>();
}

public class Patient
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int DoctorId { get; set; }
    public virtual Doctor Doctor { get; set; }
    public virtual ICollection<Appointment> Appointments { get; set; } = new List<Appointment>();
}

public class Appointment
{
    public int Id { get; set; }
    public string Description { get; set; }
    public int PatientId { get; set; }
    public virtual Patient Patient { get; set; }
}

// Ignored Entity
public class TemporaryData
{
    public int Id { get; set; }
    public string TempInfo { get; set; }
}

// Eager Loading Entity (Used only when explicitly included)
public class DoctorDetails
{
    public int DoctorId { get; set; }
    public string Specialization { get; set; }
    public int YearsOfExperience { get; set; }
}

// View Model
public class DoctorView
{
    public int DoctorId { get; set; }
    public string DoctorName { get; set; }
    public int PatientCount { get; set; }
}

// Table Function Model
public class AppointmentSummary
{
    public int PatientId { get; set; }
    public string PatientName { get; set; }
    public int AppointmentCount { get; set; }
}

// Configurations
public class DoctorConfiguration : IEntityTypeConfiguration<Doctor>
```

```
{
    public void Configure(EntityTypeBuilder<Doctor> builder)
    {
        builder.HasKey(d => d.Id);
        builder.Property(d => d.Name).IsRequired().HasMaxLength(100);
        builder.HasMany(d => d.Patients).WithOne(p => p.Doctor).HasForeignKey(p => p.DoctorId);
    }
}

public class PatientConfiguration : IEntityTypeConfiguration<Patient>
{
    public void Configure(EntityTypeBuilder<Patient> builder)
    {
        builder.HasKey(p => p.Id);
        builder.Property(p => p.Name).IsRequired().HasMaxLength(100);
        builder.HasMany(p => p.Appointments).WithOne(a => a.Patient).HasForeignKey(a => a.PatientId);
    }
}

public class AppointmentConfiguration : IEntityTypeConfiguration<Appointment>
{
    public void Configure(EntityTypeBuilder<Appointment> builder)
    {
        builder.HasKey(a => a.Id);
        builder.Property(a => a.Description).HasMaxLength(200);
    }
}

class Program
{
    static void Main()
    {
        using (var context = new MyDbContext())
        {
            // Ensure database is created
            context.Database.EnsureCreated();

            // Add a Doctor
            var doctor = new Doctor { Name = "Dr. Smith" };
            context.Doctors.Add(doctor);
            context.SaveChanges();

            // Add a Patient
            var patient = new Patient { Name = "John Doe", DoctorId = doctor.Id };
            context.Patients.Add(patient);
            context.SaveChanges();

            // Add an Appointment
            var appointment = new Appointment { Description = "Routine Checkup", PatientId = patient.Id };
            context.Appointments.Add(appointment);
            context.SaveChanges();

            // Add Temporary Data
            var tempData = new TemporaryData { TempInfo = "This is temporary info" };
```

```
            context.Add(tempData);
            context.SaveChanges();

            // Query and Display Data with Lazy Loading
            var doctors = context.Doctors.ToList();
            foreach (var doc in doctors)
            {
                Console.WriteLine($"Doctor: {doc.Name}, Patients: {doc.Patients.Count}");
            }

            // Access Temporary Data from memory no DB
            var tempInfo = context.ChangeTracker.Entries<TemporaryData>().Select(e => e.Entity.TempInfo).FirstOrD
            Console.WriteLine($"Temporary Data: {tempInfo}");

            // Query and Display View Data
            var doctorViews = context.Set<DoctorView>().ToList();
            foreach (var view in doctorViews)
            {
                Console.WriteLine($"DoctorView - ID: {view.DoctorId}, Name: {view.DoctorName}, Patients: {view.Patie
            }

            // Query and Display Function Data
            var appointmentSummaries = context.Set<AppointmentSummary>().ToList();
            foreach (var summary in appointmentSummaries)
            {
                Console.WriteLine($"Patient ID: {summary.PatientId}, Name: {summary.PatientName}, Appointments: {
            }
        }
    }
  }
}
```

What is Migrations in Entity Framework Core?

Migrations in **Entity Framework Core (EF Core)** allow you to manage and update your **database schema** based on changes in your entity models. Instead of manually writing SQL scripts to modify the database, EF Core migrations help apply changes automatically.

Why Use Migrations?

- Automatically generate database schema changes.
- Keep track of schema versions.
- Apply updates to an existing database without losing data.

How to Apply Migrations in EF Core

1. Install Packages:

```
Microsoft.EntityFrameworkCore.SqlServe
Microsoft.EntityFrameworkCore.Tools  ⇒ to add migrations in design
Microsoft.Extensions.Configuration
Microsoft.Extensions.Configuration.Json
```

2. add `json` file that contain your connection string with properties Embedded & Copy always
3. Add External configuration in your Context
4. use this command in Package Console to add migrations : Add-Migrations migration name
5. Update Data base to add Data base schema with data : Update-database
6. if you want to Remove it use : `Remove-Migration -Force`

💡 Mapping in Entity Framework Core :

1️⃣ One-to-One (1:1) Relationships

✅ Optional on Both Sides

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Address? Address { get; set; } // Optional
}

public class Address
{
    public int Id { get; set; }
    public string Street { get; set; }
    public int? PersonId { get; set; } // Optional
    public Person? Person { get; set; }
}
```

Fluent API:

```
modelBuilder.Entity<Person>()
    .HasOne(p => p.Address)
    .WithOne(a => a.Person)
    .HasForeignKey<Address>(a => a.PersonId)
    .IsRequired(false); // Optional
```

✅ Mandatory on Both Sides

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; } = null!; // Required
}

public class Address
{
    public int Id { get; set; }
    public string Street { get; set; }
    public int PersonId { get; set; } // Required
    public Person Person { get; set; } = null!; // Required
}
or mapping in same table
```

Fluent API:

```
modelBuilder.Entity<Person>()
    .HasOne(p => p.Address)
    .WithOne(a => a.Person)
    .HasForeignKey<Address>(a => a.PersonId)
    .IsRequired();
```

✅ Optional on One Side, Mandatory on the Other

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Address? Address { get; set; } // Optional
}

public class Address
{
    public int Id { get; set; }
    public string Street { get; set; }
    public int PersonId { get; set; } // Required
    public Person Person { get; set; } = null!; // Required
}
```

Fluent API:

```
modelBuilder.Entity<Person>()
    .HasOne(p => p.Address)
    .WithOne(a => a.Person)
    .HasForeignKey<Address>(a => a.PersonId)
    .IsRequired();
```

2️⃣ One-to-Many (1:N) Relationships

✅ Optional on Both Sides

```
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Book>? Books { get; set; } // Optional
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int? AuthorId { get; set; } // Optional
    public Author? Author { get; set; }
}
```

Fluent API:

```
modelBuilder.Entity<Book>()
    .HasOne(b => b.Author)
    .WithMany(a => a.Books)
    .HasForeignKey(b => b.AuthorId)
    .IsRequired(false);
```

✅ Mandatory on Both Sides

```
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Book> Books { get; set; } = new(); // Required
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; } // Required
    public Author Author { get; set; } = null!; // Required
}
```

Fluent API:

```
modelBuilder.Entity<Book>()
    .HasOne(b => b.Author)
    .WithMany(a => a.Books)
    .HasForeignKey(b => b.AuthorId)
    .IsRequired();
```

✅ Optional on One Side, Mandatory on the Other

```
public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Book>? Books { get; set; } // Optional
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; } // Required
    public Author Author { get; set; } = null!; // Required
}
```

Fluent API:

```
modelBuilder.Entity<Book>()
    .HasOne(b => b.Author)
    .WithMany(a => a.Books)
    .HasForeignKey(b => b.AuthorId)
    .IsRequired();
```

3️⃣ Many-to-Many (N:N) Relationships

✅ Optional on Both Sides

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Course>? Courses { get; set; } // Optional
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }
    public List<Student>? Students { get; set; } // Optional
}
```

Fluent API:

```
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
    .WithMany(c => c.Students)
    .UsingEntity(j => j.ToTable("StudentCourses"));
```

✅ Mandatory on Both Sides

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Course> Courses { get; set; } = new(); // Required
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }
    public List<Student> Students { get; set; } = new(); // Required
}
```

Fluent API:

```
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
```

```
    .WithMany(c => c.Students)
    .UsingEntity(j => j.ToTable("StudentCourses"));
```

✅ Optional on One Side, Mandatory on the Other

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Course>? Courses { get; set; } // Optional
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }
    public List<Student> Students { get; set; } = new(); // Required
}
```

Fluent API:

```
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
    .WithMany(c => c.Students)
    .UsingEntity(j => j.ToTable("StudentCourses"));
```

💡 **ملحوظات هامة:**

- في العلاقات **Many-to-Many** في EF Core الحديثة، يتم إنشاء الجدول الوسيط تلقائيًا بدون الحاجة إلى كيان منفصل.
- في العلاقات **One-to-One**، المفتاح الأجنبي عادة يكون في أحد الجانبين فقط.
- `IsRequired(false)` تعني أن العلاقة اختيارية (`nullable`)، بينما `IsRequired()` تجعلها إجبارية.

🚀 هل لديك أي استفسارات أو تحتاج إلى تخصيص العلاقات لحالتك؟ 😃

💡 Here are three examples of different types of relationships in Entity Framework Core:

1️⃣ One-to-One (1:1) Relationship

👉 **Example:** An **Instructor** has **one** Office, and an **Office** belongs to **one** Instructor.

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

    public int OfficeId { get; set; } // Foreign Key
    public Office Office { get; set; } // Navigation Property
}

public class Office
{
    public int Id { get; set; } // Primary Key
    public string Location { get; set; }

    public Instructor Instructor { get; set; } // Navigation Property
}

public class AppDbContext : DbContext
{
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<Office> Offices { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Instructor>()
            .HasOne(i => i.Office)
            .WithOne(o => o.Instructor)
            .HasForeignKey<Instructor>(i => i.OfficeId) // FK in Instructor
            .IsRequired();
    }
}
```

📌 **Explanation:**

- Each `Instructor` must have **exactly one** `Office`.
- `OfficeId` is a **foreign key** in `Instructor` referring to `Office`.

2️⃣ One-to-Many (1:M) Relationship

👉 **Example:** A **Department** has **many** Courses, and each **Course** belongs to **one** Department.

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

```
    public ICollection<Course> Courses { get; set; } = new List<Course>(); // Navigation Property
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }

    public int DepartmentId { get; set; } // Foreign Key
    public Department Department { get; set; } // Navigation Property
}

public class AppDbContext : DbContext
{
    public DbSet<Department> Departments { get; set; }
    public DbSet<Course> Courses { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Course>()
            .HasOne(c => c.Department)
            .WithMany(d => d.Courses)
            .HasForeignKey(c => c.DepartmentId); // FK in Course
    }
}
```

📌 **Explanation:**

- A `Department` can have **multiple Courses**.
- Each `Course` has a **foreign key** `DepartmentId` linking it to a `Department`.

3️⃣ Many-to-Many (M:N) Relationship

👉 **Example:** A **Student** can enroll in **multiple Courses**, and a **Course** can have **multiple Students**.

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

    public ICollection<Course> Courses { get; set; } = new List<Course>(); // Navigation Property
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }

    public ICollection<Student> Students { get; set; } = new List<Student>(); // Navigation Property
}

public class AppDbContext : DbContext
{
```

```
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
            .HasMany(s => s.Courses)
            .WithMany(c => c.Students)
            .UsingEntity(j => j.ToTable("StudentCourses")); // Join table
    }
}
```

📌 **Explanation:**

- **EF Core automatically creates a join table** `StudentCourses` to store `(StudentId, CourseId)`.
- No need to create a separate `Enrollment` entity manually.

Summary

Relationship Type	Example
One-to-One (1:1)	Instructor ↔ Office
One-to-Many (1:M)	Department ↔ Courses
Many-to-Many (M:N)	Students ↔ Courses

💡 If you want to add additional columns in the many-to-many relationship (`StudentCourses`), you need to create an explicit join entity instead of relying on EF Core's automatic join table.

✅ Updated Many-to-Many Relationship with Additional Columns

Example:

A `Student` can enroll in multiple `Courses`, and a `Course` can have multiple `Students`.

But now, we add **extra columns**:

- `EnrolledDate`
- `Grade`

1️⃣ Define the Join Entity (`StudentCourse`)

```
public class StudentCourse
{
    public int StudentId { get; set; }  // Composite PK + FK
    public int CourseId { get; set; }   // Composite PK + FK

    public DateTime EnrolledDate { get; set; }  // Additional Column
    public string Grade { get; set; }           // Additional Column

    public Student Student { get; set; }  // Navigation Property
    public Course Course { get; set; }    // Navigation Property
}
```

2️⃣ Update `Student` and `Course` Models

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

    public ICollection<StudentCourse> StudentCourses { get; set; } = new List<StudentCourse>();
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }

    public ICollection<StudentCourse> StudentCourses { get; set; } = new List<StudentCourse>();
}
```

3️⃣ Configure Relationships in `AppDbContext`

```
public class AppDbContext : DbContext
{
```

```
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
    public DbSet<StudentCourse> StudentCourses { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<StudentCourse>()
            .HasKey(sc => new { sc.StudentId, sc.CourseId }); // Composite PK

        modelBuilder.Entity<StudentCourse>()
            .HasOne(sc => sc.Student)
            .WithMany(s => s.StudentCourses)
            .HasForeignKey(sc => sc.StudentId);

        modelBuilder.Entity<StudentCourse>()
            .HasOne(sc => sc.Course)
            .WithMany(c => c.StudentCourses)
            .HasForeignKey(sc => sc.CourseId);
    }
}
```

4️⃣ Seeding Example (Adding Data)

```
using (var context = new AppDbContext())
{
    var student1 = new Student { Name = "John Doe" };
    var course1 = new Course { Title = "Mathematics" };

    var enrollment = new StudentCourse
    {
        Student = student1,
        Course = course1,
        EnrolledDate = DateTime.Now,
        Grade = "A"
    };

    context.Students.Add(student1);
    context.Courses.Add(course1);
    context.StudentCourses.Add(enrollment);

    context.SaveChanges();
}
```

📌 Summary

Column	Stored In
`StudentId`	`StudentCourses` (FK + PK)
`CourseId`	`StudentCourses` (FK + PK)
`EnrolledDate`	`StudentCourses` (Extra Column)
`Grade`	`StudentCourses` (Extra Column)

How can Merage Migrations ?

💡 Steps to Merge Migrations 🏗️

1️⃣ Delete Old Migrations 🗑️

- Go to your `Migrations` folder 📂
- **Delete all migration files (except the first one, if needed)**
- Also, remove their entries from the `__EFMigrationsHistory` table in the **database** 🛢️

2️⃣ Update the Model Snapshot 🔄

- Open `YourDbContextModelSnapshot.cs` 📝
- Ensure it reflects your **current models** ✅
- If there are old references, **remove them manually**! 🛠️

3️⃣ Rebuild and Create a New Migration 🚀

- **Rebuild** your project ⚙️ to ensure models are up-to-date
- Create a **new migration file** manually by copying necessary changes ✍️

4️⃣ Auto-Apply Migration (No Commands!) ⚡

Instead of running commands, add this `Program.cs`:

```
using (var scope = app.Services.CreateScope())
{
   var dbContext = scope.ServiceProvider.
   GetRequiredService<ApplicationDbContext>();

   var migration = dbContext.Database.GetPendingMigrations();

   if (migration.Any())
      {
   dbContext.Database.Migrate(); // 🛠️ Auto-applies latest migration!
   }
}

//Code in Console App

      static void Main(string[] args)
      {
         using (var dbContext = new AppDbContext())
         {

            var migration = dbContext.Database.GetPendingMigrations();
            //GetPendingMigrations retrun unapplyed Migrations

            if (migration.Any())
            {
               dbContext.Database.Migrate(); // 🛠️ Auto-applies latest migration!
```

```
        }
    }
}
```

💥 **Now, when your app runs, the latest migration will be applied automatically!**

Without using Command `Update-database` every time , now working in your app and and your several migrations and in end when you run app it update database 🎯

📌 How to Map a Composite Attribute?

EF Core allows you to define a **complex type** (composite attribute) as an **owned entity** inside another entity. The data of this owned entity will be **stored inside the same table** as the main entity.

✅ Step-by-Step Implementation

1️⃣ Define the Composite Attribute as a Class

First, create a separate class for the composite attribute.

```
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}
```

✔️ **This class represents a composite attribute** because it has multiple properties (`Street`, `City`, `PostalCode`).

2️⃣ Use the Composite Attribute in an Entity

Now, let's create an entity (`Patient`) that includes this composite attribute.

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

    public Address Address { get; set; } // ✅ Composite Attribute (Owned Entity)
}
```

✔️ The `Patient` entity has an `Address` property, which acts as a composite attribute.

3️⃣ Configure Owned Entity in `DbContext`

Now, configure `Address` as an **Owned Entity** in `OnModelCreating()`.

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Patient> Patients { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Patient>()
            .OwnsOne(p => p.Address,a=>
            {
            a.property(x=>x.Street).HascolumnName("Street");
             a.property(x=>x.City).HascolumnName("City");
              a.property(x=>x.PostalCode).HascolumnName("PostalCode");
            }); // 🔥 Mapping Composite Attribute!
    }
```

```
}
//we use HascolumnName to add owned configurations
// not use default Convention like Address_Street
```

✔️ The `.OwnsOne(p ⇒ p.Address)` tells EF Core that `Address` **is NOT a separate table** but will be stored **inside the** `Patients` **table**.

4️⃣ Result in Database (Single Table)

When you create the database, the `Patients` table will look like this:

Id	Name	Street	City	PostalCode
1	Ali	123 Main St	Cairo	12345
2	Ahmed	456 Elm St	Giza	67890

✔️ Notice that the `Address` properties are **merged into the** `Patients` **table** instead of creating a separate table.

✔️ EF Core will **automatically map the data** from the database into the `Address` property of the `Patient` entity.

How can apply Value Conversion in Entity Framework Core ?

🎯 Value Conversion in EF Core

Value Conversion in **Entity Framework Core** allows you to transform data when storing it in the database and converting it back when retrieving it. This is useful when you want to **store data in a specific format** that differs from how you use it in your C# code.

📌 Why Use Value Converters?

✅ Store **Enums as Strings** instead of numbers.

✅ Store `booleans` **as integers** (`0` or `1`).

✅ Convert **encrypted values** before storing.

✅ Example 1: Convert Enum to String

By default, EF Core stores enums as integers. You can **convert enums to strings** using `HasConversion()`.

🔷 Define an Enum

```
public enum Gender
{
    Male,
    Female
}
```

🔷 Use the Enum in an Entity

```
public class Patient
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Gender Gender { get; set; } // ✅ Enum Property
}
```

🔷 Apply the Value Converter in `DbContext`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Patient>()
        .Property(p => p.Gender)
        .HasConversion<string>(); // 🎯 Convert Enum to String
}
```

🛠️ Result in Database

Id	Name	Gender
1	Ali	Male
2	Ahmed	Female

✔️ **Now `Gender` is stored as "Male"/"Female" instead of `0` or `1`.**

✅ Example 2: Convert Boolean to Integer

By default, EF Core stores `bool` as `BIT` **(0 or 1)** in SQL Server. But if you want to store it as `0` or `100`, use **value conversion**.

🔷 Define an Entity

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool IsActive { get; set; } // ✅ Boolean Property
}
```

🔷 Apply the Value Converter in `DbContext`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .Property(u => u.IsActive)
        .HasConversion(
            v => v ? 100 : 0, // ✅ Convert `true` → `100`, `false` → `0`
            v => v == 100     // ✅ Convert `100` → `true`, `0` → `false`
        );
}
```

🛠️ Result in Database

Id	Name	IsActive
1	Sara	100
2	Ahmed	0

✔ **Now `IsActive` is stored as `100` instead of `1`, and `0` instead of `false`.**

✅ Example 3: Store JSON Data in a Column

You can store **complex objects as JSON** instead of using related tables.

🔷 Define a Complex Type

```
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

🔷 Use It in an Entity

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

```
    public Address Address { get; set; } // ✅ Complex Object
}
```

🔥 Conclusion

- **Value Converters** allow you to store data in a different format than your C# model.
- **Common Uses:**
  - ✅ Convert **Enums** to Strings
  - ✅ Convert **Booleans** to Integers
  - ✅ Encrypt/Decrypt values before storing

💡 How can In Entity Framework Core inheritance mapping strategies determine how related entities are stored in the database?

The three primary inheritance strategies are:

1. **Table Per Hierarchy (TPH)**
2. **Table Per Type (TPT)**
3. **Table Per Concrete Type (TPC)**

Each approach has **advantages, disadvantages, and use cases**, depending on **performance, normalization, and query complexity**.

1️⃣ Table Per Hierarchy (TPH)

👉 **Concept:**

- Uses **a single table** to store **all entities in the hierarchy**.
- Differentiates entity types using a **Discriminator column**.

📌 Example

```
public abstract class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Doctor : Employee
{
    public string Specialization { get; set; }
}

public class Nurse : Employee
{
    public string Department { get; set; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
            .HasDiscriminator<string>("EmployeeType") // Discriminator column
            .HasValue<Doctor>("Doctor")
            .HasValue<Nurse>("Nurse");
    }
}
```

```
//if you want to apply default you must put a DbSet for childs if you wanted to //use convientionthen conumn will named as Discriminator
```

📌 Database Schema (Single Table: `Employees`)

Id	Name	EmployeeType	Specialization	Department
1	Alice	Doctor	Cardiology	NULL
2	Bob	Nurse	NULL	Pediatrics
3	John	Doctor	Neurology	NULL

📌 Pros & Cons

✅ **Pros:**

- **Performance is high** (Single table, no joins).
- **Easier to query** (All data in one place).
- **Simple to implement**.

❌ **Cons:**

- **Lots of NULL values** (If properties vary between types).
- **Less normalization** (Data integrity issues).
- **Difficult schema changes** (Adding a new type requires adding columns).

📌 Best Use Cases

- **Performance-sensitive applications**.
- **Few subtypes with shared properties**.
- **Scenarios where frequent queries on all subtypes are needed**.

2️⃣ Table Per Type (TPT)

👉 **Concept:**

- Each entity in the hierarchy has **its own table**.
- The base entity's **primary key** is **used as a foreign key** in derived tables.

📌 Example

```
public abstract class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Doctor : Employee
{
    public string Specialization { get; set; }
}

public class Nurse : Employee
{
    public string Department { get; set; }
```

```
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Doctor> Doctors { get; set; }
    public DbSet<Nurse> Nurses { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {

        modelBuilder.Entity<Employee>().UseTptMappingStrategy();

        modelBuilder.Entity<Doctor>().ToTable("Doctors");
        modelBuilder.Entity<Nurse>().ToTable("Nurses");
    }
}
```

📌 Database Schema

`Employees` **Table (Base Table)**

Id	Name
1	Alice
2	Bob

`Doctors` **Table**

Id	Specialization
1	Cardiology

`Nurses` **Table**

Id	Department
2	Pediatrics

📌 Pros & Cons

✅ **Pros:**

- **Well-normalized structure** (Better data integrity).
- **No NULL values** (Each table has only relevant fields).
- **Easier schema evolution** (Adding new subtypes is cleaner).

❌ **Cons:**

- **Performance impact** (Requires joins to get complete entity data).
- **More complex queries** (Fetching all employees requires multiple table joins).

📌 Best Use Cases

- **Highly structured applications where data integrity is important**.
- **When different subtypes have vastly different properties**.
- **When queries are usually made per subtype rather than across all types**.

3️⃣ Table Per Concrete Type (TPC)

👉 **Concept:**

- Each **concrete class** has its **own table**, but **without a shared base table**.
- No need for a **discriminator column**.

📌 Example

```
public abstract class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Doctor : Employee
{
    public string Specialization { get; set; }
}

public class Nurse : Employee
{
    public string Department { get; set; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Doctor> Doctors { get; set; }
    public DbSet<Nurse> Nurses { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>().UseTpcMappingStrategy();
        modelBuilder.Entity<Doctor>().ToTable("Doctors");
        modelBuilder.Entity<Nurse>().ToTable("Nurses");
    }
}
```

📌 Database Schema

`Doctors` **Table**

Id	Name	Specialization
1	Alice	Cardiology

`Nurses` **Table**

Id	Name	Department
2	Bob	Pediatrics

📌 Pros & Cons

✅ **Pros:**

- **No need for joins** (Each type is stored separately).
- **Cleaner schema** (Only relevant fields exist in each table).

- **Fast queries for individual types**.

❌ **Cons:**

- **Data duplication** (If many common fields exist in all types).
- **No shared constraints** (No central `Employees` table).
- **More complex queries across different types**.

📌 Best Use Cases

- **When each entity has significantly different fields**.
- **When querying individual types separately is common**.
- **When data duplication is acceptable for performance reasons**.

📊 Summary Table

Feature	TPH (Table Per Hierarchy)	TPT (Table Per Type)	TPC (Table Per Concrete Type)
Tables Count	**1 Table**	**Multiple Tables**	**Multiple Tables**
Normalization	**Low (NULLs present)**	**High (No NULLs, foreign keys used)**	**Moderate (No NULLs, but data duplication)**
Performance	**High (Single table, no joins)**	**Medium (Requires joins)**	**High (No joins, but data duplication)**
Query Complexity	**Simple**	**Complex (Joins required)**	**Simple for specific types, Complex for hierarchy queries**
Best for	**Few types, fast queries**	**Structured, well-normalized data**	**Independent tables, overlapping types**

🚀 Which One Should You Choose?

✅ **Use TPH if:**

- Performance is critical.
- Few subtypes with shared properties.

✅ **Use TPT if:**

- Data integrity and normalization are important.
- You expect to add new subtypes frequently.

✅ **Use TPC if:**

- Entities have very different structures.
- Querying individual types is more common than querying across all types.

🔷 Create & Drop APIs in EF Core

EF Core provides built-in methods to create and drop databases dynamically:

API Method	Description
`EnsureCreated()`	Creates the database if it does not exist. **Does not use migrations.**
`EnsureDeleted()`	Deletes the database if it exists.
`Migrate()`	Applies all pending migrations to update the database schema.

🔷 How to Use Create & Drop APIs in EF Core

You can use these methods inside the `ApplicationDbContext` class or in your application startup.

1️⃣ `EnsureCreated()` – Create Database Without Migrations

```
using (var context = new ApplicationDbContext())
{
    context.Database.EnsureCreated();
}
```

✅ **Use Case:**

- Creates the database only if it **does not exist**.
- Bypasses migrations and directly generates tables based on entity models.
- Ideal for **quick prototyping** or **testing**.

❌ **Limitations:**

- If you add new entities later, it won't update the schema.
- Not recommended for production.

2️⃣ `EnsureDeleted()` – Drop Database If Exists

```
using (var context = new ApplicationDbContext())
{
    context.Database.EnsureDeleted();
}
```

✅ **Use Case:**

- Deletes the database if it exists.
- Useful during **unit testing** or **restarting from scratch**.

3️⃣ Combining `EnsureDeleted() & EnsureCreated()`

If you want to **reset the database** every time you run the application:

```
using (var context = new ApplicationDbContext())
{
    context.Database.EnsureDeleted();
    context.Database.EnsureCreated();
}
```

✅ **Use Case:**

- Useful for **recreating the database** during development.
- Ensures fresh data every time.

4️⃣ Using Migrate() – Preferred for Production

Instead of `EnsureCreated()`, use `Migrate()` in production:

```
using (var context = new ApplicationDbContext())
{
   context.Database.Migrate();
}
```

✅ **Use Case:**

- Applies all pending **migrations** to keep the database updated.
- Recommended for **real applications** instead of `EnsureCreated()`.

🔷 When to Use Which API?

Scenario	Recommended API
Quick testing / temporary database	`EnsureCreated()`
Reset database every run (testing)	`EnsureDeleted()` + `EnsureCreated()`
Production (update schema safely)	`Migrate()`

🔷 Conclusion

- `EnsureCreated()` : Quick database setup (skips migrations).
- `EnsureDeleted()` : Deletes database if exists.
- `Migrate()` : Properly applies migrations for production.

Let me know if you need further clarification! 🚀

✅ Best Practices for Seeding Data in `Program.cs` in EF Core

Seeding data in `Program.cs` allows you to insert, update, or delete initial data dynamically at application startup. This method **avoids the limitations of** `HasData()` and is preferred for scenarios where:

- Data may change over time.
- You need dynamic values like `DateTime.Now`.
- You want to avoid migration-based seeding.

🔷 1️⃣ Steps to Seed Data in `Program.cs`

1. **Ensure Dependency Injection is set up for** `ApplicationDbContext`.
2. **Use a scoped service to seed data inside** `Program.cs`.
3. **Check if data already exists before inserting new data.**
4. **Call** `SaveChanges()` **after adding or modifying data.**

🔷 2️⃣ Example: Seeding Data in `Program.cs`

👨‍⚕️ Scenario:

You are managing a hospital database and need to insert initial departments dynamically at application startup.

```
using (var scope = app.Services.CreateScope())
{
    var context = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

    try
    {
        context.Database
        .EnsureCreated(); // Ensures DB is created before seeding.

        // Seed Departments if not already present.
        if (!context.Departments.Any())
        {
            context.Departments.AddRange(
                new Department { Name = "Ophthalmology" },
                new Department { Name = "Pediatrics" },
                new Department { Name = "Operations" }
            );
            context.SaveChanges();
        }

        // Seed Doctors (Example of related entity)
        if (!context.Doctors.Any())
        {
            context.Doctors.AddRange(
                new Doctor { Name = "Dr. John", Specialization = "Eye Specialist", DepartmentId = 1 },
                new Doctor { Name = "Dr. Sarah", Specialization = "Pediatrician", DepartmentId = 2 }
            );
            context.SaveChanges();
        }
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Database seeding failed: {ex.Message}");
    }
}
```

🔷 3️⃣ Explanation

✅ `EnsureCreated()` → Ensures the database is created before inserting data.

✅ `context.Departments.Any()` → Prevents duplicate records.

✅ `SaveChanges()` → Saves changes after adding entities.

✅ **Using** `try-catch` → Catches errors if the database is not available.

🔷 4️⃣ Why is `Program.cs` Seeding Better Than `HasData()`?

Feature	`HasData()`	`Program.cs` Seeding
Can modify data later	❌ No	✅ Yes
Supports `DateTime.Now` and dynamic values	❌ No	✅ Yes
Requires migrations	✅ Yes	❌ No
Handles relationships easily	❌ No	✅ Yes
Checks for duplicates	❌ No	✅ Yes

Limitation of use `HasData()` : can not use with owned Entity

if you use it you will customization your migration

🔷 Final Thoughts (Best Practices)

✔️ **Always check for existing data (`.Any()`)** to prevent duplicates.

✔️ **Use `EnsureCreated()` only for development**, not production (use migrations instead).

✔️ **Use a `try-catch`** block to handle errors in case of database connection issues.

✔️ **For complex seeding**, create a separate `DbInitializer` class instead of putting everything in `Program.cs`.

🚀 **Now your app will always have initial data on startup!** 🚀

What happen in this Code ?

```
static void Main(string[] args)
{
    using (var dbContext = new AppDbContext())
    {

        var courses = dbContext.Courses;

        foreach (var item in courses)
        {
            Console.WriteLine(item.CourseName);
        }
    }
}
```

1. `var courses = dbContext.Courses;`

- Here, `courses` is assigned the value of `dbContext.Courses`.
- `dbContext.Courses` is an **IQueryable<Course>**, meaning that no database query is executed at this point.
- Instead, an **EF Core query object** is created, representing a SQL query that **has not been executed yet** (deferred execution).

---

2. `foreach (var item in courses)`

- The `foreach` loop starts iterating over `courses`.
- Since `courses` is an `IQueryable<Course>`, **Entity Framework Core translates the LINQ expression into a SQL query** and sends it to the database.
- The database executes the query and returns the results **as a stream of objects (** `IEnumerable<Course>` **)**, meaning that objects are fetched **one by one as needed**, rather than loading everything into memory at once.

🔥 ⚡ Understanding Query Evaluation in EF Core (Server vs. Client Evaluation) 🚀

🧐 What's the Problem?

When using Entity Framework Core (EF Core) to fetch data from the database, queries can be executed either **on the server (Server-side Evaluation)** or **on the application (Client-side Evaluation)**. The difference **greatly affects performance** and **application speed**.

1️⃣ Deferred Execution ⏳

🔷 When you write a query like this:

```
var courses = context.Courses.Where(c ⇒ c.Level > 1);
```

It **does not execute immediately**. Instead, it waits until the data is **actually requested** (e.g., using `ToList()` or `foreach`).

✅ **This improves performance since the query runs only when needed.**

2️⃣ Server-side Evaluation 🖥️⚡

🔷 If a query can be **translated into SQL**, it will be executed **directly in the database**.

✅ **This is the best and fastest option because only the required data is retrieved.**

🔷 Example:

```
var courses = context.Courses
    .Where(c ⇒ c.Level > 1)
    .ToList(); // Executed on the server ✅
```

🔷 Generated SQL:

```
SELECT * FROM Courses WHERE Level > 1;
```

👆 **The database filters the results and returns only what's needed.**

3️⃣ Client-side Evaluation 🖥️🐌

🔷 If the query contains **C# code that cannot be translated to SQL**, **all data is fetched first, then filtered in the application!**

❌ **This is bad for performance because it retrieves a large amount of unnecessary data.**

🔷 Example:

```
var courses = context.Courses
    .ToList() // ❌ Fetches all records
    .Where(c ⇒ CalculateLevel(c.Level)); // Filtering happens in C# ❌
```

🔷 **The problem?** 😱

- **All data** is fetched from the database 🏋️.
- Takes **longer time** 🐌.
- Uses **more memory** instead of leveraging SQL's efficiency.

4️⃣ How to Avoid Client-side Evaluation? 🚀

✅ **Use functions that EF Core can translate into SQL**, like:

```
var courses = context.Courses
    .Where(c => c.Name.Contains("Math")) // ✅ SQL supports Contains
    .ToList();
```

✅ **Avoid calling** `ToList()` **before** `Where()`

❌ **Wrong:**

```
var courses = context.Courses.ToList().Where(c => c.Level > 1);
```

✅ **Correct:**

```
var courses = context.Courses.Where(c => c.Level > 1).ToList();
```

✅ **Use** `EF.Functions.Like()` **instead of** `StringComparison`

```
var users = context.Users.Where(u => EF.Functions.Like(u.Name, "%John%")).ToList();
```

5️⃣ How to Detect Client-side Evaluation? 🔍

🔭 **Enable query logging:**

```
optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information);
```

👀 **If you see a SQL query in Console fetching all data without a** `WHERE` **clause, you have a problem!**

⬅️ END Summary: How to Write Fast Queries?

✅ **Use queries that can be translated into SQL.**

✅ **Don't call** `ToList()` **before** `Where()`**.**

✅ **Use** `EF.Functions.Like()` **instead of** `String.Contains()`**.**

✅ **Enable query logging to check where execution happens.**

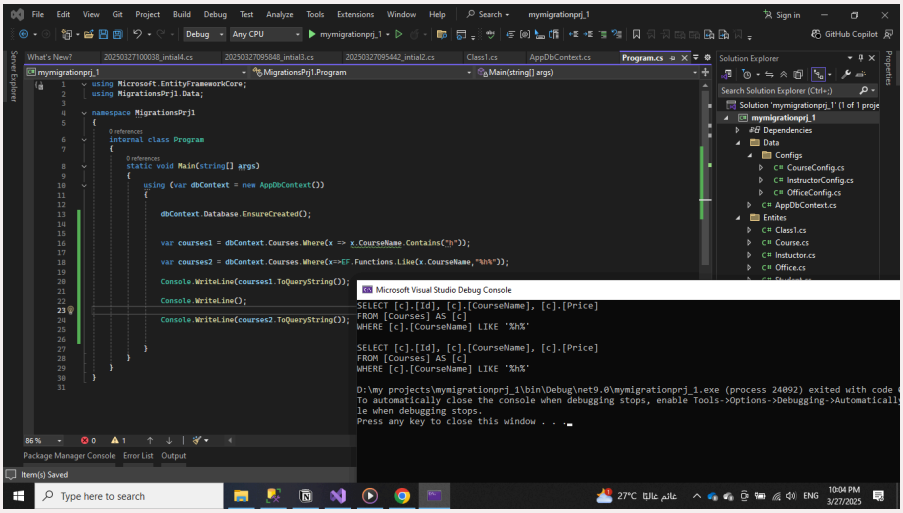
💡 **The more filtering you do in SQL instead of C#, the faster your application will run!** 🚀

Compare between two queries?

```
var courses1 = dbContext.Courses.
Where(x ⇒ x.CourseName.Contains("h"));

var courses2 = dbContext.Courses.
Where(x⇒EF.Functions.Like(x.CourseName,"%h%"));
```

here two queries is the same work in server-side because two converted in same SQL query




```
SELECT [c].[Id], [c].[CourseName], [c].[Price]
FROM [Courses] AS [c]
WHERE [c].[CourseName] LIKE '%h%';
```

but all time contains not work in server-side ,

🔍 Tracking vs. NoTracking in EF Core 🚀

Entity Framework Core (EF Core) has two modes for querying data:

🔷 **Tracking Queries** (default)

🔷 **NoTracking Queries** (explicitly set)

1️⃣ Tracking Queries (`AsTracking()`)

EF Core **tracks** changes made to the retrieved entities in the **DbContext**.

✅ Best for **modifications (Update, Delete, etc.)**.

Example:

```
var course = dbContext.Courses.FirstOrDefault(x ⇒ x.Id == 1);
course.CourseName = "Updated Name";
dbContext.SaveChanges(); // ✅ EF Core detects changes & updates DB!
```

SQL Generated:

```
SELECT TOP(1) [c].[Id], [c].[CourseName], [c].[Price]
FROM [Courses] AS [c]
WHERE [c].[Id] = 1;
```

💡 **EF Core keeps the entity in memory and tracks changes!**

2️⃣ NoTracking Queries (`AsNoTracking()`)

EF Core **does NOT track** the entity, meaning **changes won't be saved automatically**.

✅ Best for **read-only operations** (e.g., reports, performance-critical queries).

Example:

```
var course = dbContext.Courses.AsNoTracking().FirstOrDefault(x ⇒ x.Id == 1);
course.CourseName = "Updated Name";
dbContext.SaveChanges(); // ❌ No effect! Changes are NOT tracked.
```

SQL Generated (Same as Before, but No Tracking in Memory):

```
SELECT TOP(1) [c].[Id], [c].[CourseName], [c].[Price]
FROM [Courses] AS [c]
WHERE [c].[Id] = 1;
```

💡 **EF Core fetches data but does NOT track changes in memory!**

⚡ Key Differences:

Feature	Tracking ✅	NoTracking 🚀
Tracks Changes?	✅ Yes	❌ No
Saves Changes?	✅ Yes (auto-detects changes)	❌ No (must explicitly attach entity)
Performance	🐢 Slower (more memory usage)	⚡ Faster (better for read-heavy queries)

Best Use Cases	**CRUD operations (Insert, Update, Delete)**	**Read-only queries (Reports, Large Data Reads, API Responses, etc.)**

🎯 When to Use What?

🔷 Use **Tracking** when modifying data.

🔷 Use **NoTracking** for faster read operations (e.g., displaying lists, reports).

🚀 **Tip:** If you need to update a **NoTracking entity**, attach it manually:

```
dbContext.Courses.Update(course);
dbContext.SaveChanges();
```

🔍 **TL;DR:** `AsTracking()` keeps entities in memory & detects changes, while `AsNoTracking()` fetches data without tracking for better performance! 🚀

👌

How to Change Tracking Behavior in Entity Framework Core?

You can change tracking behavior in different ways:

1. Set Default Tracking Behavior in `DbContext`

Disable tracking for all queries by default:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
   optionsBuilder.UseSqlServer("Your_Connection_String")
                 .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
}
```

2. Change Tracking for a Specific Entity

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   modelBuilder.Entity<Doctor>()
      .Metadata.SetQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
}
```

3. Change Tracking for a Specific Query

- **Disable tracking for a single query:**

```
var doctors = context.Doctors.AsNoTracking().ToList();
```

- **Enable tracking for a query (if tracking is disabled by default):**

```
var doctor = context.Doctors.AsTracking().FirstOrDefault(d ⇒ d.Id == 1);
```

When to Use `NoTracking`?

✔️ When reading data without modifying it (better performance).

✔️ For reports or read-only operations.

When Not to Use `NoTracking`?

❌ When updating or deleting records (because EF won't track changes).

❌ When using **lazy loading** (it won't work with `NoTracking`).

How can load Related Data?

by use :

1. Lazy Loading
2. Eager Loading
3. Explicit Loading

Difference Between Lazy Loading and Eager Loading in Entity Framework Core

Lazy Loading and Eager Loading are two different ways to load related data in **Entity Framework Core**.

1️⃣ Lazy Loading (`Lazy Loading`)

📌 **Definition**: Related data is loaded **only when accessed** for the first time.

📌 **How it works**: Uses **proxy objects** or **manual loading** (`NavigationProperty.Load()`).

Example (Lazy Loading using Virtual Navigation Properties)

- install package

```
Install-Package Microsoft.EntityFrameworkCore.Proxies
```

use `lazyloading` in `DbContext` and use virtual with navigation property

```
protected override void OnConfiguring
(DbContextOptionsBuilder optionsBuilder)
{

      optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer("Your_Connection_String");

}

public class Doctor
{
   public int Id { get; set; }
   public string Name { get; set; }

   // Lazy loading enabled with 'virtual'
   public virtual List<Patient> Patients { get; set; }
}
```

> If you fetch a Doctor, its Patients list will be loaded only when you access it for the first time:

```
var doctor = context.Doctors.First();
var patients = doctor.Patients; // Lazy-loaded here
```

✅ Pros of Lazy Loading

✔️ Loads only when needed (reduces initial query cost).

✔️ Useful for large datasets when related data isn't always required.

❌ Cons of Lazy Loading

❌ Multiple database queries (can cause **N+1 problem**).

❌ Needs additional configuration (`virtual` properties or proxies).

2️⃣ Eager Loading (`Include`, `ThenInclude`)

📌 **Definition**: Loads related data **immediately** when querying the main entity.

📌 **How it works**: Uses `.Include()` and `.ThenInclude()` to load relationships.

Example (Eager Loading with `.Include()`)

```
var doctor = context.Doctors
    .Include(d => d.Patients) // Load Patients with Doctor
    .First();
```

Now, when you access `doctor.Patients`, **the data is already loaded**.

Example (Eager Loading with `.ThenInclude()`)

If `Patient` has a `List<MedicalRecords>`, you can load it like this:

```
var doctor = context.Doctors
    .Include(d => d.Patients)
        .ThenInclude(p => p.MedicalRecords) // Load MedicalRecords inside Patients
    .First();
```

✅ Pros of Eager Loading

✔️ Loads everything in **one query** (better performance for multiple relations).

✔️ Avoids the **N+1 query problem**.

✔️ Works well with complex object graphs.

❌ Cons of Eager Loading

❌ Loads all related data **even if not needed** (higher memory usage).

❌ Can lead to **large SQL queries** if too many `.Include()` calls are used.

🚀 Which One Should You Use?

- **Use Lazy Loading** ✅ When you **don't always need related data**.
- **Use Eager Loading** ✅ When you **always need related data** (avoids extra queries).

🔵 What is Explicit Loading in Entity Framework Core?

Explicit Loading means manually loading related data **when needed**, instead of using **Eager Loading (** `.Include()` **)** or **Lazy Loading (** `virtual` **)**.

With **Explicit Loading**, you load the related data **on demand**, using the `Entry().Collection()` or `Entry().Reference()` methods.

🛠️ How to Apply Explicit Loading?

1. `Collection()` → Used for **one-to-many** or **many-to-many** relationships.
2. `Reference()` → Used for **one-to-one** or **many-to-one** relationships.

📌 Example: Loading Related Patients for a Doctor (One-to-Many)

🔷 Step 1: Define Your Entities

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

    public virtual List<Patient> Patients { get; set; } // Navigation Property
}

public class Patient
{
    public int Id { get; set; }
    public string FullName { get; set; }

    public int DoctorId { get; set; }
    public virtual Doctor Doctor { get; set; }
}
```

📌 Example: Loading Related Patients for a Doctor (One-to-Many)

```
using (var context = new AppDbContext())
{
    var doctor = context.Doctors.FirstOrDefault(); // Load only doctor (without patients)

    if (doctor != null)
    {
        // ✅ Explicitly load the related patients using .Query() for filtering
        var patients = context.Entry(doctor)
                              .Collection(d => d.Patients)
                              .Query()
                              .Where(p => p.Id > 5)  // Apply filtering
                              .OrderBy(p => p.FullName)
                              .ToList(); // Execute query

        Console.WriteLine($"Doctor: {doctor.Name}, Patients Count: {patients.Count}");
```

```
    }
}
```

📌 **Why `.Query()` Instead of `.Load()`?**

✔️ **Better performance**: Only loads the required records.

✔️ **Supports filtering & sorting**: `.Where()`, `.OrderBy()`.

✔️ **Deferred execution**: The query runs **only when needed**.

📌 Example: Loading a Single Related Object (One-to-One / Many-to-One)

```
using (var context = new AppDbContext())
{
    var patient = context.Patients.FirstOrDefault(); // Load only patient (without doctor)

    if (patient != null)
    {
        // ✅ Explicitly load the related doctor using .Query()
        var doctor = context.Entry(patient)
                            .Reference(p => p.Doctor)
                            .Query()
                            .FirstOrDefault(); // Execute query

        Console.WriteLine($"Patient: {patient.FullName}, Doctor: {doctor?.Name}");
    }
}
```

📌 **Why `.Query()` Here?**

✔️ Allows **filtering** before execution.

✔️ Fetches **only needed data** instead of full navigation property.

🚀 Conclusion: Always Use `.Query()` If Possible!

🔷 `.Load()` loads **everything** into memory, can not apply Filtering with it

🔷 `.Query()` loads **only what is needed**.

✅ **Best practice:** Use `.Query()` for **better performance** and **control over queries**. 🚀

🎯 When to Use Explicit Loading?

✅ When you **don't want to always load related data** (like Eager Loading).

✅ When **Lazy Loading is not enabled** but you still want to load related data on demand.

✅ When you need **more control** over database queries and performance.

⚠️

📊 Comparison Between Eager Loading & Explicit Loading

Feature	**Eager Loading (`Include()`)**	**Explicit Loading (`Entry().Load()`)**
How it Works	Loads related data **immediately** with the main entity.	Loads related data **manually** when needed.
When Data is Loaded	At the time of the first query.	After the main entity is loaded.
Code Example	`context.Doctors.Include(d => d.Patients).ToList();`	`context.Entry(doctor).Collection(d => d.Patients).Load();`
Control Over Data	Loads everything unless you use filtering (`.Where()`).	You decide **when** and **what** to load.
Performance	May load **unnecessary** data, increasing query size.	Loads **only** when needed, reducing initial load time.
Filtering	`context.Doctors.Include(d => d.Patients.Where(p => p.Age > 18))` (possible but limited).	Use `.Query().Where().Load()` for precise filtering.
Lazy Loading Needed?	❌ No, it works independently.	❌ No, it works manually.
Best Use Case	When you **always** need related data **immediately**.	When you **sometimes** need related data and want control.

🚀 **Conclusion:**

- **Eager Loading** (`Include()`) is best when you **always need** related data upfront.
- **Explicit Loading** (`Entry().Load()`) is better when you want **manual control** and only load data **when required**.

🗡️ Comparison Between Projection (`Select`) and Eager Loading (`Include`) in Entity Framework Core

Criteria	Projection (`Select`)	Eager Loading (`Include`)
Description	Retrieves **only specific fields** using `Select()` .	Retrieves **the entire entity and its related data** using `.Include()` .
Performance	**Better performance** as it fetches only the required fields.	Can be **slower** due to loading extra data.
Use of `Include()`	Does not use `.Include()` , manually selects required fields.	Uses `.Include()` to fetch related data.
Number of Queries	**Single query** with selected fields.	**Single or multiple queries** (depending on relationships).
Data Size	**Minimal data size**, fetching only selected fields.	Can be **large**, as it loads all related data.
When Full Entity is Needed	**Not suitable** if you need the complete entity object.	**Suitable** when you need the full entity with related data.
When Partial Data is Needed	**Best choice** when you need only specific data fields.	**Not efficient** as it loads extra unused data.
Code Example	`csharp context.Courses .Select(c => new { c.Id, c.Name, Sections = c.Sections.Select(s => new { s.Id, s.Name }) }) .ToList();`	`csharp context.Courses .Include(c => c.Sections) .ToList();`

When to Use Each?

- ✅ **Use** `Projection` when you need **only specific data** and not the full entity.
- ✅ **Use** `Eager Loading` when you need **to load all related data** upfront without additional queries.

🚀 **Conclusion**: If you want to optimize performance and reduce data size, use `Select()` **(Projection)**. If you need to fetch the full entity with related data, use `.Include()` **(Eager Loading)**.

⚠️

What is a Splitting Query in Entity Framework Core?

A **splitting query** in Entity Framework Core is a technique used to fetch related entities using **multiple SQL queries instead of a single complex JOIN query**. This helps improve performance by reducing redundant data retrieval and avoiding large result sets that may cause performance bottlenecks.

How Does It Work?

- Normally, when you use `.Include()`, EF Core generates a **single SQL query with multiple JOINs** to fetch the main entity and its related data. This can lead to **data duplication and inefficiency**.
- With **split queries**, EF Core **executes multiple queries**, each fetching different parts of the data separately, then **combines** them in memory.

Example Without Splitting Query (Single Query with JOINs)

```
var courses = context.Courses
    .Include(c ⇒ c.Sections) // Loads related sections in a single query
    .ToList();
```

✅ **Pros**: Single database call.

❌ **Cons**: Can cause **redundant data loading** due to JOINs.

Example Using a Splitting Query

```
var courses = context.Courses
    .AsSplitQuery() // Enables split queries
    .Include(c ⇒ c.Sections)
    .ToList();
```

✅ **Pros**: Avoids data duplication and reduces query complexity.

❌ **Cons**: Multiple database calls may increase **network latency**.

When to Use Splitting Queries?

✔️ When you have **one-to-many** or **many-to-many** relationships.

✔️ When JOINs cause **data duplication** and performance issues.

✔️ When retrieving **large amounts of related data**.

Conclusion

Splitting queries (`AsSplitQuery()`) can help improve performance by avoiding redundant data loading. However, use it carefully because it results in **multiple database calls**, which might impact performance if not managed properly. 🚀

🔑 Q: How can I wisely use `AsSplitQuery()` in EF Core to improve performance?

A: First, check the SQL logs or use a profiler to analyze the generated queries. If you notice a lot of duplicated data being loaded, it may impact performance. Run the query directly in the database to confirm. If excessive duplication occurs, use `AsSplitQuery()` to split the query into multiple smaller ones, reducing redundant data retrieval. Finally, compare performance before and after applying it to ensure optimization. 🚀

💡 What is Paging ?

Paging is a technique used to split a large set of data into smaller, more manageable chunks or pages. Instead of fetching all the rows from a database table, you retrieve only a subset of rows at a time. This improves performance and user experience, especially when displaying data in tables or lists.

✅ How Paging Works

- You specify:
  - **Page Number**: The current page you want to fetch.
  - **Page Size**: The number of items to display per page.
- EF Core uses `.Skip()` and `.Take()` methods to implement paging.

📌 Implementing Paging in EF Core

Suppose you have an entity named `Patient` and you want to implement paging when fetching patients from the database.

```
int pageNumber = 1;  // The page number you want to display
int pageSize = 10;   // The number of records per page

var patients = await _context.Patients
    .OrderBy(p => p.Name)  // Always order the data before paging
    .Skip((pageNumber - 1) * pageSize)
    .Take(pageSize)
    .ToListAsync();
```

🔑 Explanation

- `.OrderBy()`: It's important to order your data before applying paging to maintain consistency across pages.
- `.Skip()`: Skips a certain number of rows based on the page size and current page number.
- `.Take()`: Fetches the specified number of rows (page size).

🔍 Raw SQL Queries in EF Core

EF Core supports executing raw SQL queries directly against the database. This is useful when you need complex queries that are difficult to express using LINQ or when you want to leverage database-specific features.

EF Core provides three main methods for executing raw SQL queries:

---

1. `FromSqlRaw()`

This method executes a **raw SQL query** with **plain text**. You must be careful with SQL injection risks when using it.

```
var sql3 = context.Courses.FromSqlRaw("Select * FROM Courses where Id = 1" )
.FirstOrDefault();

//here it will be Select * FROM Courses where Id = 1

//must use sqlparameter to safe from sql injection
var sqlparm = new SqlParameter("@courseid", 1);


var sql3 = context.Courses.FromSqlRaw("Select * FROM Courses where Id = @courseid", sqlparm).FirstOr
Default();

//Select * FROM Courses where Id = @courseid
```

- **No parameterization**, so it's unsafe if you concatenate user input directly.
- Use this when you are sure the SQL string is safe.

---

2. `FromSqlInterpolated()`

This method allows **parameterized SQL queries** using **string interpolation** (`$""`). It is safer and protects against SQL injection.

```
var sql2 = context.Courses.FromSqlInterpolated($"Select * FROM Courses where Id ={1}").FirstOrDefault();

//"Select * FROM Courses where @p0
```

- **Safe against SQL injection** because parameters are **automatically escaped**.
- Recommended when you need to use parameters from variables.

---

3. `FromSql()`

- In **EF Core 2.x**, `FromSql()` was the only method available for raw SQL queries.
- It has been replaced by `FromSqlRaw()` and `FromSqlInterpolated()` in **EF Core 3.0+**.
- **Do not use it if you are on EF Core 3.0 or higher.**

🔑 Differences

Method	Safe Against SQL Injection	Recommended Use Case

`FromSqlRaw()`	❌ No	Static queries with no parameters.
`FromSqlInterpolated()`	✅ Yes	Queries with dynamic parameters.
`FromSql()`	❌ Deprecated	Only for older EF Core versions.

📌 Example with Your Project

If you want to **fetch doctors from a specific department**:

```
var sql2 = context.Courses.FromSql($"Select * FROM Courses where Id ={1}")
.FirstOrDefault();

//"Select * FROM Courses where @p0);
```

📌 Important Tips

1. **Always use `FromSqlInterpolated()` for dynamic queries** to avoid SQL injection.
2. `FromSqlRaw()` **is useful for completely static queries**.
3. **Make sure your raw SQL matches your model properties.** The columns in the SQL query must match the properties of the entity type.

🛠️ How can use SQL Server objects like stored procedures, scalar functions, and table-valued functions in an EF Core ?

✅ 1. Using Stored Procedures

Creating a Stored Procedure in SQL Server

```
CREATE PROCEDURE GetPatientsByDepartment
   @DepartmentId INT
AS
BEGIN
   SELECT * FROM Patients WHERE DepartmentId = @DepartmentId
END
```

Calling a Stored Procedure in EF Core

In your `ApplicationDbContext` class:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
   public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
   {
   }
 protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
     modelBuilder.Entity<Patient>(x⇒x.HasNoKey());
   }
   public DbSet<Patient> Patients { get; set; }

   // Example of calling a stored procedure
   public async Task<List<Patient>> GetPatientsByDepartmentAsync(int departmentId)
   {
      return await Patients
         .FromSqlRaw("EXEC GetPatientsByDepartment @DepartmentId = {0}", departmentId)
         .ToListAsync();
   }
}
```

✅ 2. Using Scalar Functions

Creating a Scalar Function in SQL Server

```
CREATE FUNCTION GetDoctorNameById (@DoctorId INT)
RETURNS NVARCHAR(100)
AS
BEGIN
   DECLARE @DoctorName NVARCHAR(100)
   SELECT @DoctorName = Name FROM Doctors WHERE Id = @DoctorId
```

```
    RETURN @DoctorName
END
```

Calling a Scalar Function in EF Core

You need to define a **model-level query** in your `ApplicationDbContext`:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {
    }

    public DbSet<Doctor> Doctors { get; set; }

    [DbFunction("GetDoctorNameById", "dbo")] // Assuming function is in the dbo schema
    public static string GetDoctorNameById(int doctorId) => throw new NotImplementedException();
}
```

And you can use it like this:

```
var doctorName = ApplicationDbContext.GetDoctorNameById(1);
Console.WriteLine(doctorName);
```

To use a **Table-Valued Function (TVF)** with an expression (`FromExpression`) in EF Core,

---

✅ 1. Create a Table-Valued Function in SQL Server

For example, let's say you have a function that returns doctors from a specific department:

```
CREATE FUNCTION dbo.GetDoctorsByDepartment (@DepartmentId INT)
RETURNS TABLE
AS
RETURN
(
    SELECT Id, Name, Specialization
    FROM Doctors
    WHERE DepartmentId = @DepartmentId
);
```

✅ 2. Define a Model Class to Map the Result

Create a class that matches the result of your TVF:

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

```
    public string Specialization { get; set; }
}
```

✅ 3. Configure Your DbContext

In your `ApplicationDbContext`:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {
    }

    public DbSet<DoctorInfo> DoctorInfos { get; set; }  // For mapping TVF results

    // Define your TVF as an expression
    public IQueryable<DoctorInfo> GetDoctorsByDepartment(int departmentId)
        => FromExpression(() => GetDoctorsByDepartment(departmentId));

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Configure the table-valued function mapping
        modelBuilder
            .HasDbFunction(typeof(ApplicationDbContext).GetMethod(nameof(GetDoctorsByDepartment)))
            .HasName("GetDoctorsByDepartment")
            .HasSchema("dbo");  // Adjust schema if different

        // Configure the entity type for the TVF result
        modelBuilder.Entity<DoctorInfo>().HasNoKey().ToTable("DoctorInfos", t => t.ExcludeFromMigrations());
    }
}
```

✅ 4. Using the TVF in Your Application

```
using (var context = new ApplicationDbContext(options))
{
    var doctors = context.GetDoctorsByDepartment(1).ToList();

    foreach (var doctor in doctors)
    {
        Console.WriteLine($"{doctor.Name} - {doctor.Specialization}");
    }
}
```

📌 Explanation

1. `FromExpression()` is used for defining functions that can be **transformed into SQL by EF Core**.

2. In the `OnModelCreating()` method:
    - `HasDbFunction()` maps your C# method to the SQL function.
    - `HasName()` specifies the name of the TVF in the database.
    - `HasSchema()` specifies the schema where the function resides (`dbo` by default).
    - `HasNoKey()` indicates that the result is not tracked as a regular entity.
    - `ToTable("DoctorInfos", t => t.ExcludeFromMigrations())` prevents migrations from trying to create a table for this TVF.

To use a SQL View in EF Core, you need to map it as a `DbSet` in your `DbContext` and configure it using `OnModelCreating()`.

✅ 1. Creating a View in SQL Server

Let's say you have a SQL View that combines doctor and department data:

```
CREATE VIEW dbo.DoctorDepartmentView AS
SELECT
    d.Id AS DoctorId,
    d.Name AS DoctorName,
    d.Specialization,
    dep.Id AS DepartmentId,
    dep.Name AS DepartmentName
FROM Doctors d
JOIN Departments dep ON d.DepartmentId = dep.Id;
```

✅ 2. Define a Model Class to Map the View

Create a model class that matches the structure of your view:

```
public class DoctorDepartmentView
{
    public int DoctorId { get; set; }
    public string DoctorName { get; set; }
    public string Specialization { get; set; }
    public int DepartmentId { get; set; }
    public string DepartmentName { get; set; }
}
```

✅ 3. Configure Your `DbContext`

In your `ApplicationDbContext`:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {
    }
```

```
    // Define DbSet for the view
    public DbSet<DoctorDepartmentView> DoctorDepartmentViews { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Configure the view mapping
        modelBuilder
            .Entity<DoctorDepartmentView>()
            .HasNoKey()  // Views do not have primary keys
            .ToView("DoctorDepartmentView", "dbo"); // Specify the view name and schema

        modelBuilder.Entity<DoctorDepartmentView>()
        .ToTable("DoctorDepartmentViews", t => t.ExcludeFromMigrations());
         // Avoid EF Core trying to create it
    }
}
```

✅ 4. Using the View in Your Application

```
using (var context = new ApplicationDbContext(options))
{
    var result = context.DoctorDepartmentViews.ToList();

    foreach (var item in result)
    {
        Console.WriteLine($"{item.DoctorName} ({item.Specialization}) - Department: {item.DepartmentNam
e}");
    }
}
```

📌 Explanation

1. `DbSet<DoctorDepartmentView>` : This is how you access your view via EF Core.
2. `HasNoKey()` : Since views don't have primary keys, you need to specify this.
3. `ToView("DoctorDepartmentView", "dbo")` : Maps the model to the SQL View by name.
4. `ExcludeFromMigrations()` : Prevents EF Core from trying to create this as a table when running migrations.

✅ Global Filters in EF Core

Global Filters in EF Core are used to automatically filter rows returned by a query for a particular entity type. They are particularly useful for implementing **soft deletes, multi-tenancy, or user-specific data filtering**.

🔍 How to Define a Global Filter

You define global filters in your `DbContext`'s `OnModelCreating()` method.

📌 Example: Soft Delete Implementation

Suppose you have an `Entity` class with a `IsDeleted` property.

```
public class Doctor
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool IsDeleted { get; set; } // Soft delete property
}
```

📌 Configuring the Global Filter

In your `DbContext`:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {
    }

    public DbSet<Doctor> Doctors { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Global filter to exclude soft-deleted entities
        modelBuilder.Entity<Doctor>().HasQueryFilter(d => !d.IsDeleted);
    }
}
```

🔥 How It Works

When you query the `Doctors` DbSet:

```
using (var context = new ApplicationDbContext(options))
{
    var doctors = context.Doctors.ToList(); // Automatically excludes soft-deleted doctors
}
```

The generated SQL will **always include the filter**:

```
SELECT * FROM Doctors WHERE IsDeleted = 0;
```

📌 Disabling Global Filters Temporarily

If you need to **include soft-deleted entities** in a particular query, you can use `.IgnoreQueryFilters()`:

```
var allDoctors = context.Doctors.IgnoreQueryFilters().ToList(); // Ignores the global filter
```

🌟 What is `Attach()` in EF Core?

The `Attach()` method in Entity Framework Core is used to **start tracking an existing entity without marking it for insertion, update, or deletion**. It tells EF Core:

> "Hey, this entity already exists in the database, just track it for changes if I decide to modify or delete it later."

📌 How `Attach()` Works

```
context.Attach(entity);
```

- **State:** `Unchanged` (by default) → The entity is considered to exist in the database without pending changes.
- **SQL Command Generated:** None, unless you change the entity's state explicitly (`Modified`, `Deleted`, etc.) or modify its properties and call `SaveChanges()`.
- **Common Use Cases:** Setting relationships, updating existing data without fetching it, deleting entities by their primary keys, etc.

🔥 Example Usage

✅ Scenario 1: Update an Existing Entity

Suppose you have a `Doctor` entity already in the database, and you want to update its name without fetching it first.

```
var doctor = new Doctor { Id = 1, Name = "Dr. Updated" };
context.Attach(doctor);
context.Entry(doctor).State = EntityState.Modified; // Mark as modified
context.SaveChanges(); // Generates an UPDATE SQL statement
```

✅ Scenario 2: Setting Relationships Without Fetching

If you have a `Patient` that should be linked to an existing `Doctor`.

```
var doctor = new Doctor { Id = 1 }; // Only providing the Id
context.Attach(doctor); // Track the doctor without fetching from the database

var patient = new Patient { Name = "John", Doctor = doctor };
context.Patients.Add(patient); // Relates the patient to the tracked doctor
context.SaveChanges(); // Saves both Patient and the relationship
```

✅ Scenario 3: Deleting an Entity Without Fetching

Instead of fetching an entity to delete it:

```
var doctor = new Doctor { Id = 1 };
context.Attach(doctor);
context.Remove(doctor); // Mark it as Deleted
context.SaveChanges(); // Generates a DELETE SQL statement
```

📌 Best Practices for Using `Attach()`

1. Avoid Unnecessary Database Queries

Instead of:

```
var doctor = context.Doctors.Find(1);
doctor.Name = "Updated Name";
context.SaveChanges();
```

Use:

```
var doctor = new Doctor { Id = 1, Name = "Updated Name" };
context.Attach(doctor);
context.Entry(doctor).State = EntityState.Modified;
context.SaveChanges();
```

- 🔥 **Benefit:** Saves a round-trip to the database, improving performance.

---

2. Use `Attach()` for Relationships

When adding related entities, use `Attach()` to avoid unnecessary loading.

```
var doctor = new Doctor { Id = 1 };
context.Attach(doctor);

var patient = new Patient { Name = "New Patient", Doctor = doctor };
context.Patients.Add(patient);
context.SaveChanges();
```

---

3. Combine with `Entry()` for Updates

Marking an entity as modified after attaching:

```
var doctor = new Doctor { Id = 1, Name = "New Name" };
context.Attach(doctor);
context.Entry(doctor).Property(d => d.Name).IsModified = true;
context.SaveChanges();
```

---

4. Avoid Using `Attach()` for New Entities

If the entity is not in the database, `Attach()` will cause an error if you try to modify or delete it. Instead, use `Add()` or `Update()`.

📌 When NOT to Use `Attach()`

1. 🚫 When adding new entities - use `Add()` instead.
2. 🚫 When fetching data from the database with tracking enabled (`ToList()`, `Find()`, etc.) - it's already being tracked.
3. 🚫 When you need to ensure the entity exists in the database - `Attach()` assumes the entity exists without checking.

📌 Summary

- `Attach()` is great for working with existing entities that you don't want to fetch from the database.
- It improves performance by avoiding unnecessary queries.
- Always combine with `Entry()` if you intend to mark entities as modified or deleted.
- Use it **mostly for updates, deletions, and establishing relationships**.

✅ What Are Delete Behaviors in Entity Framework Core?

`DeleteBehavior` is a setting in Entity Framework Core that **controls what happens to related entities when a parent entity is deleted**. This is important when you have relationships defined between entities using `foreign keys`.

🔥 Types of Delete Behaviors

1. Cascade (`DeleteBehavior.Cascade`)

- **Effect:** When you delete a parent entity, all related child entities are automatically deleted.
- **Use Case:** When deleting a `Doctor`, all related `Messages` or `Appointments` should also be deleted.

```
modelBuilder.Entity<Doctor>()
    .HasMany(d ⇒ d.Messages)
    .WithOne(m ⇒ m.Doctor)
    .OnDelete(DeleteBehavior.Cascade);
```

🔍 Example:

If you delete a `Doctor`, all `Messages` linked to that doctor are also deleted.

2. Restrict (`DeleteBehavior.Restrict`)

- **Effect:** Prevents deleting a parent entity if related child entities exist.
- **Use Case:** When you want to ensure that certain data is not deleted if it has related data.

```
modelBuilder.Entity<Doctor>()
    .HasMany(d ⇒ d.Patients)
    .WithOne(p ⇒ p.Doctor)
    .OnDelete(DeleteBehavior.Restrict);
```

🔍 Example:

If you try to delete a `Doctor` who has existing `Patients`, EF Core will throw an error.

3. SetNull (`DeleteBehavior.SetNull`)

- **Effect:** When the parent is deleted, the `foreign key` in the related child entity is set to `null`.
- **Use Case:** When deleting a `Doctor`, you may want to keep the `Patient` records but remove their association with that `Doctor`.

```
modelBuilder.Entity<Patient>()
    .HasOne(p ⇒ p.Doctor)
    .WithMany(d ⇒ d.Patients)
    .OnDelete(DeleteBehavior.SetNull);
```

🔍 Example:

If you delete a `Doctor`, the `DoctorId` in the related `Patient` records will be set to `null`.

4. NoAction (`DeleteBehavior.NoAction`)

- **Effect:** EF Core doesn't enforce any specific action on delete. The database itself will decide what to do (e.g., throw an error if foreign key constraints are violated).
- **Use Case:** Useful when the database is managing foreign key constraints instead of EF Core.

```
modelBuilder.Entity<Doctor>()
    .HasMany(d => d.Messages)
    .WithOne(m => m.Doctor)
    .OnDelete(DeleteBehavior.NoAction);
```

🔍 Example:

If you delete a `Doctor`, and there are related `Messages`, the deletion will fail unless you manually delete the messages or handle the constraint.

---

5. ClientSetNull (`DeleteBehavior.ClientSetNull`)

- **Effect:** Similar to `SetNull`, but only applies to tracking of entities on the client side (EF Core). It won't affect the database itself.
- **Use Case:** When you want to handle null references in your C# code but not enforce the same in the database.

```
modelBuilder.Entity<Patient>()
    .HasOne(p => p.Doctor)
    .WithMany(d => d.Patients)
    .OnDelete(DeleteBehavior.ClientSetNull);
```

📌 Best Practices for Your Project

In your hospital project, you might use:

- `DeleteBehavior.Cascade`: If you want to delete all messages related to a doctor when the doctor is deleted.
- `DeleteBehavior.Restrict`: If you want to prevent deleting a `Doctor` if there are patients still assigned to them.
- `DeleteBehavior.SetNull`: If you want to keep `Patient` records but make them independent of a deleted `Doctor`.

🌟 Example Configuration for Your Project

```
modelBuilder.Entity<Doctor>()
    .HasMany(d => d.Messages)
    .WithOne(m => m.Doctor)
    .OnDelete(DeleteBehavior.Cascade); // Deletes all messages when a doctor is deleted.

modelBuilder.Entity<Patient>()
    .HasOne(p => p.Doctor)
    .WithMany(d => d.Patients)
    .OnDelete(DeleteBehavior.SetNull); // Sets DoctorId to null if the doctor is deleted.

modelBuilder.Entity<Doctor>()
    .HasMany(d => d.Patients)
    .WithOne(p => p.Doctor)
    .OnDelete(DeleteBehavior.Restrict); // Prevents deleting a doctor if they have patients.
```

✅ What Is Bulking in EF Core?

Bulking in EF Core refers to performing **batch operations** (Insert, Update, Delete) on **multiple rows at once** with a **single database command**, rather than iterating over each entity and issuing separate SQL statements for each operation.

This is done to **enhance performance and efficiency**, especially when dealing with large amounts of data.

🔥 Why Bulking Is Important

- 🔄 **Faster Execution:** One SQL command instead of many.
- 💾 **Reduced Memory Usage:** No need to load entities into memory.
- ⚡ **Improved Performance:** Fewer round-trips to the database.
- 📊 **Better Scalability:** Suitable for large-scale applications.

To use Bilking install Package:

```
Install-Package EFCore.BulkExtensions
```

📌 Examples of Bulking Operations

1. Bulk Insert (High Performance Insertion)

In normal EF Core operations:

```
foreach (var patient in patients)
{
   context.Patients.Add(patient);
}
await context.SaveChangesAsync();
```

This will generate **multiple** `INSERT` **statements** (one for each patient), which is slow.

✅ **Using Bulk Insert (via third-party libraries like EFCore.BulkExtensions):**

```
await context.BulkInsertAsync(patients);
```

- Generates a **single** `INSERT` **statement** for all patients.
- Extremely fast compared to the standard approach.

---

2. Bulk Update (High Performance Update)

Instead of this:

```
foreach (var doctor in doctors)
{
   doctor.IsActive = true;
}
await context.SaveChangesAsync();
```

Which sends an `UPDATE` **statement for each doctor**.

✅ **Using** `ExecuteUpdate()` **(EF Core 7.0+)**:

```
await context.Doctors
    .Where(d => d.DepartmentId == departmentId)
    .ExecuteUpdateAsync(d => d.SetProperty(x => x.IsActive, x => true));
```

- Generates a **single** `UPDATE` **statement**.
- No tracking required, improving performance.

---

3. Bulk Delete (High Performance Deletion)

Instead of this:

```
foreach (var message in messagesToDelete)
{
    context.Messages.Remove(message);
}
await context.SaveChangesAsync();
```

This will generate **multiple** `DELETE` **statements**.

✅ **Using** `ExecuteDelete()` **(EF Core 7.0+)**:

```
await context.Messages
    .Where(m => m.CreatedAt < DateTime.Now.AddMonths(-6))
    .ExecuteDeleteAsync();
```

- Generates a **single** `DELETE` **statement**.
- Highly efficient, especially for large datasets.

📚 Using Third-Party Libraries for Bulking

EF Core doesn't have built-in support for bulk operations other than `ExecuteUpdate()` and `ExecuteDelete()` in version 7.0+. However, libraries like:

- `EFCore.BulkExtensions`
- `Z.EntityFramework.Extensions`

Provide **additional functionalities** for bulk insert, update, delete, and merge operations with better performance.

📈 Best Practices for Using Bulking

1. **Use** `ExecuteUpdate()` **and** `ExecuteDelete()` **for bulk modifications in EF Core 7.0+.**
2. **Use third-party libraries for older versions of EF Core or when needing more complex bulk operations.**
3. **Use bulking for performance-critical parts of your application, like updating patient records or deleting old messages.**
4. **Always test bulk operations in a development environment before deploying to production.**

✅ What is Soft Delete?

Soft Delete is a technique where **data is marked as deleted instead of being physically removed from the database**.

This is achieved by **adding a column (`IsDeleted`, for example) to the table**, and setting its value to `true` when the data is considered deleted.

🎯 Why Use Soft Delete?

1. ✅ **Data Retention:** You can recover data later if needed.
2. ✅ **Tracking:** You can keep track of when and who deleted the data.
3. ✅ **Safety:** Minimizes the risk of losing important data by mistake.

📌 How to Apply Soft Delete in EF Core

1. Modify the Entity (`Patient`)

```
public class Patient
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Department { get; set; }
    public bool IsDeleted { get; set; } = false; // Column for Soft Delete
}
```

2. Configure `DbContext` to Use Soft Delete (Global Filter)

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public DbSet<Patient> Patients { get; set; }

    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Patient>().ToTable("Patients");

        // Apply a global filter to exclude soft-deleted data
        modelBuilder.Entity<Patient>().HasQueryFilter(p => !p.IsDeleted);
    }
}
```

3. Apply Soft Delete Instead of Deleting Data (Soft Delete Implementation)

```
public async Task SoftDeletePatient(ApplicationDbContext context, int patientId)
{
    var patient = await context.Patients.FindAsync(patientId);
    if (patient != null)
    {
        patient.IsDeleted = true; // Marking as soft deleted
        await context.SaveChangesAsync(); // Save changes to the database
    }
}
```

4. Retrieve Data Including Deleted Records (`Ignore Query Filters`)

```
public async Task<List<Patient>> GetAllPatientsIncludingDeleted(ApplicationDbContext context)
{
    return await context.Patients
                        .IgnoreQueryFilters() // Ignores the global filter to fetch all data
                        .ToListAsync();
}
```

📌 Explanation

1. 🔑 We used `HasQueryFilter()` to apply a global filter to the table, ignoring records where `IsDeleted = true`.
2. 🔑 When using `IgnoreQueryFilters()`, it bypasses global filters to retrieve all records, including the soft-deleted ones.
3. 🔑 The deletion (`Soft Delete`) is simply updating the `IsDeleted` value to `true` instead of actually deleting the record.

🌟 Example Usage

```
var patient = new Patient { Name = "Khaled", Age = 35, Department = "Ophthalmology" };
context.Patients.Add(patient);
await context.SaveChangesAsync();

// Soft Delete the patient
await SoftDeletePatient(context, patient.Id);

// Retrieve all patients (Only those with IsDeleted = false)
var patients = await context.Patients.ToListAsync(); // Khaled will NOT be included

// Retrieve all patients including deleted ones
var allPatients = await GetAllPatientsIncludingDeleted(context); // Khaled will be included
```

🎉 Advantages of Soft Delete

- Easy data recovery without requiring a backup.
- Control over data visibility based on `IsDeleted`.
- Improved safety by protecting data from unintended deletion.

💡 How can apply Soft Delete to a specific entity only using an interceptor?

✅ 1. Define `ISoftDelete` Interface

```
public interface ISoftDelete
{
    bool IsDeleted { get; set; }
}
```

✅ 2. Define Your Entity (e.g., `Patient`)

```
public class Patient : ISoftDelete
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool IsDeleted { get; set; } = false;
}
```

✅ 3. Create `SoftDeleteInterceptor`

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;

public class SoftDeleteInterceptor : SaveChangesInterceptor
{
    public override InterceptionResult<int> SavingChanges(DbContextEventData eventData, InterceptionResult
<int> result)
    {
        var context = eventData.Context;
        if (context == null) return result;

        // Apply soft delete only to the Patient entity
        foreach (var entry in context.ChangeTracker.Entries<Patient>())
        {
            if (entry.State == EntityState.Deleted)
            {
                entry.State = EntityState.Modified;
                entry.Entity.IsDeleted = true;
            }
        }

        return base.SavingChanges(eventData, result);
    }
}
```

✅ 4. Apply Interceptor in Your `DbContext`

```
using Microsoft.EntityFrameworkCore;
```

```
public class ApplicationDbContext : DbContext
{
    private readonly SoftDeleteInterceptor _softDeleteInterceptor;

    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options, SoftDeleteInterceptor soft
DeleteInterceptor)
        : base(options)
    {
        _softDeleteInterceptor = softDeleteInterceptor;
    }

    public DbSet<Patient> Patients { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.AddInterceptors(_softDeleteInterceptor);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Apply global filter to hide soft-deleted Patients
        modelBuilder.Entity<Patient>().HasQueryFilter(p => !p.IsDeleted);
    }
}
```

✅ 5. Usage Example

```
using var context = new ApplicationDbContext(options, new SoftDeleteInterceptor());

// Adding a patient
var patient = new Patient { Name = "Khaled" };
context.Patients.Add(patient);
await context.SaveChangesAsync();

// Soft deleting a patient
context.Patients.Remove(patient);
await context.SaveChangesAsync();

// Fetching all patients (only non-deleted ones will be returned)
var patients = await context.Patients.ToListAsync();
```

🔑 Explanation

1. **Apply Soft Delete only to** `Patient` **entity** by checking the type in the interceptor.
2. **Global filter** ensures that soft-deleted patients are hidden from queries.
3. Simple and clean code with minimal changes to your application.

✅ What is a Transaction?

A **transaction** is a sequence of one or more operations that are **executed as a single logical unit of work**. It ensures that either all the operations are **completed successfully** or none of them are. This is often referred to as the **ACID properties**:

1. **Atomicity**: All operations succeed or none of them are applied.
2. **Consistency**: The database is transformed from one valid state to another valid state.
3. **Isolation**: Transactions are executed independently without interference.
4. **Durability**: Once a transaction is committed, it remains even if the system crashes.

📌 When to Use Transactions

- When you are performing **multiple related database operations** that must all succeed or fail as a unit.
- Example: Transferring money between two bank accounts.

📖 How to Use Transactions in EF Core (Best Practices)

EF Core supports **transactions** via the `IDbContextTransaction` interface. Here's how you can implement them properly.

🔑 1. Using Explicit Transactions (Recommended Approach)

```
using (var transaction = await _context.Database.BeginTransactionAsync())
{
    try
    {
        var senderAccount = await _context.BankAccounts.SingleOrDefaultAsync(a => a.AccountNumber == "12
345");
        var receiverAccount = await _context.BankAccounts.SingleOrDefaultAsync(a => a.AccountNumber == "6
7890");

        if (senderAccount == null || receiverAccount == null)
            throw new Exception("Invalid account numbers");

        decimal transferAmount = 500m;

        // Check if sender has enough balance
        if (senderAccount.Balance < transferAmount)
            throw new Exception("Insufficient funds");

        // Deduct amount from sender
        senderAccount.Balance -= transferAmount;
        _context.BankAccounts.Update(senderAccount);
        await _context.SaveChangesAsync();

        // Add amount to receiver
        receiverAccount.Balance += transferAmount;
        _context.BankAccounts.Update(receiverAccount);
        await _context.SaveChangesAsync();

        // Commit the transaction if everything is successful
        await transaction.CommitAsync();
```

```
    }
    catch (Exception)
    {
        // Rollback transaction if an error occurs
        await transaction.RollbackAsync();
        throw;
    }
}
```

🔑 2. Using `TransactionScope` (For Multiple DbContexts)

If you need to perform operations across **multiple** `DbContext` **instances**:

```
using System.Transactions;

using (var scope = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
{
    var senderAccount = await _context1.BankAccounts.SingleOrDefaultAsync(a => a.AccountNumber == "1234
5");
    var receiverAccount = await _context2.BankAccounts.SingleOrDefaultAsync(a => a.AccountNumber == "67
890");

    if (senderAccount == null || receiverAccount == null)
        throw new Exception("Invalid account numbers");

    decimal transferAmount = 1000m;

    // Deduct from sender
    senderAccount.Balance -= transferAmount;
    _context1.BankAccounts.Update(senderAccount);
    await _context1.SaveChangesAsync();

    // Add to receiver
    receiverAccount.Balance += transferAmount;
    _context2.BankAccounts.Update(receiverAccount);
    await _context2.SaveChangesAsync();

    // Complete the transaction
    scope.Complete();
}
```

> Note: TransactionScope requires TransactionScopeAsyncFlowOption.Enabled for asynchronous code.

🔑 3. Implicit Transactions (Not Recommended for Complex Scenarios)

If you are only working with a **single** `DbContext` **instance**, EF Core will use implicit transactions by default for `SaveChanges()` if it detects multiple operations.

```
await _context.SaveChangesAsync();
```

This is **suitable for simple operations**, but when dealing with **multiple** `DbContext` **instances or complex operations**, you should use **explicit transactions**.

💡 Best Practices

1. **Use Explicit Transactions:**

   Always prefer `Database.BeginTransaction()` when dealing with related operations. This gives you complete control over committing or rolling back changes.

2. **Keep Transactions Short:**

   Minimize the code executed inside a transaction to reduce contention and improve performance.

3. **Handle Exceptions Properly:**

   Always use `try-catch` blocks to ensure rollback on failure.

4. **Avoid Long-Running Transactions:**

   Don't keep transactions open across multiple requests or user interactions.

5. **Use Async Methods:**

   Always prefer `BeginTransactionAsync()`, `CommitAsync()`, and `RollbackAsync()` for **non-blocking operations**.

📌 In a Banking System

In a banking system, you might use transactions when:

- **Transferring money** between accounts.
- **Processing payments** where multiple operations are involved (e.g., debiting, crediting, logging).
- **Handling batch processing** of multiple related operations.

✅ What is Rollback to Savepoint?

It's a way to **undo part of a transaction** to a specific point (savepoint) without discarding the entire transaction. Useful when you want to retry or skip a failed operation without starting over.

📖 Use Cases for Savepoints in Businesses

1. **Banking Systems:**

   While transferring funds between multiple accounts, you may want to rollback a specific account operation without affecting the entire transaction. For example, if transferring money from **Account A** to **Account B** and **Account C**, and the transfer to **Account C** fails, you can rollback to the point after transferring to **Account B** instead of starting over.

2. **E-commerce Orders:**

   When processing a large order with multiple items, if updating inventory for one item fails, you can rollback that item's change without discarding the whole order.

3. **Reservation Systems:**

   Booking multiple services (flight, hotel, car rental) within a single transaction. If the flight booking fails, rollback only the flight reservation but keep the successful hotel and car bookings.

🔑 Implementation in EF Core

```
using (var transaction = await _context.Database.BeginTransactionAsync())
{
    try
    {
        var accountA = new BankAccount { AccountNumber = "123", Balance = 5000 };
        _context.BankAccounts.Add(accountA);
        await _context.SaveChangesAsync();

        await transaction.CreateSavepointAsync("SavepointA");

        var accountB = new BankAccount { AccountNumber = "456", Balance = 3000 };
        _context.BankAccounts.Add(accountB);
        await _context.SaveChangesAsync();

        await transaction.CreateSavepointAsync("SavepointB");

        var accountC = await _context.BankAccounts.FindAsync("999");
        if (accountC == null)
        {
            await transaction.RollbackToSavepointAsync("SavepointB");
        }

        await transaction.CommitAsync();
    }
    catch
    {
        await transaction.RollbackAsync();
        throw;
```

```
    }
}
```

💡 **Best Practice:**

Use savepoints to **retry failed operations** or **skip problematic steps** without starting over.

🚀 Let’s Speed Up Performance

Top 13 Mistakes Developers Make in EF Core

🗣️ 1. Not Using Indexes

When you run queries using Entity Framework Core (EF Core), the way those queries are executed in the database can greatly impact performance. If the columns you frequently search or sort by (`WHERE` or `ORDER BY`) don't have indexes, the database will have to **scan the entire table** to find the matching rows. This process is called a **table scan**, and it can be **very slow** when dealing with large tables.

📌 What is an Index?

An index is like a **sorted list or a directory**. Instead of searching through every row in the table, the database uses the index to quickly find the relevant data.

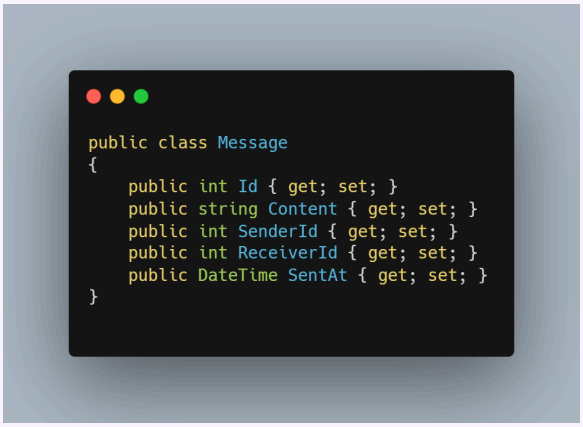
👉 Think of it like searching for a word in a dictionary. If the words weren't sorted alphabetically, you'd have to read through every word until you find the one you want. But because dictionaries are indexed (sorted), you can find the word almost instantly.

💡 Why Indexes Matter

- ✅ Faster searches
- ✅ Quicker sorting
- ✅ Improved overall performance

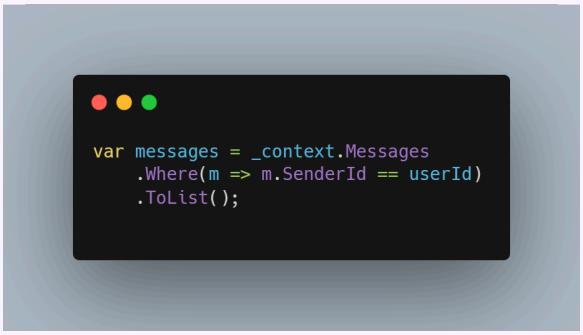
🔍 Example in EF Core (C#)

Imagine you have a `Messages` table where you frequently search for messages sent by a specific user (`SenderId`), like this:



```
public class Message
{
    public int Id { get; set; }
    public string Content { get; set; }
    public int SenderId { get; set; }
    public int ReceiverId { get; set; }
    public DateTime SentAt { get; set; }
}
```

And your queries often look like this:



```
var messages = _context.Messages
    .Where(m => m.SenderId == userId)
    .ToList();
```

If `SenderId` is not indexed, the database will **scan all rows** in the `Messages` table to find matches. This can be very slow if there are thousands or millions of rows.

🔨 Solution: Adding an Index

To optimize this, you can create an index on the `SenderId` column in your `DbContext` using Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Message>()
        .HasIndex(m => m.SenderId);
}
```

Now, when you run the same query:

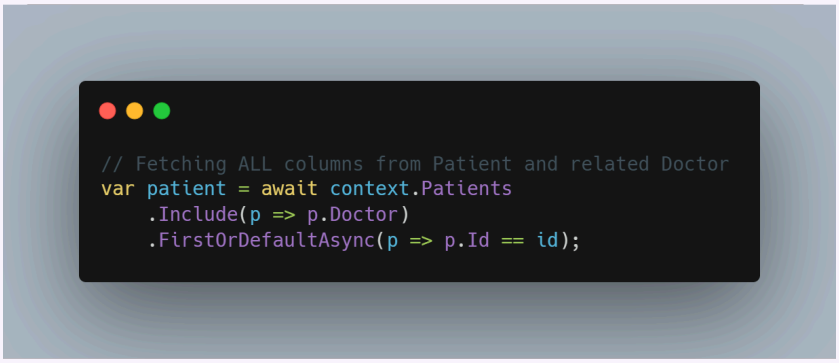
The database will **use the index** to quickly locate all rows with the matching `SenderId`, making the query much faster.

🗣️ 2. Not Using Projections

When you fetch data from the database, if you don't specify what you need, EF Core **loads everything**. This includes all columns and related data, which makes your query **slow and uses more memory**.

❌ The Wrong Way (Fetching Everything)

Imagine you have a `Patient` table and a related `Doctor` table. You just want the patient's name and their doctor's name, but you write:

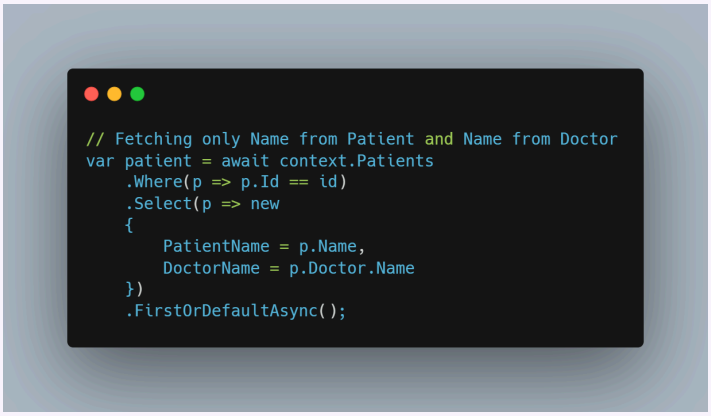


```
// Fetching ALL columns from Patient and related Doctor
var patient = await context.Patients
    .Include(p => p.Doctor)
    .FirstOrDefaultAsync(p => p.Id == id);
```

This loads **all columns** from `Patient` and **all data** from `Doctor`. Wasteful!

✅ The Right Way (Fetching Only What You Need)

Instead, you should **select only the data you need** using `.Select()` like this:



```
// Fetching only Name from Patient and Name from Doctor
var patient = await context.Patients
    .Where(p => p.Id == id)
    .Select(p => new
    {
        PatientName = p.Name,
        DoctorName = p.Doctor.Name
    })
    .FirstOrDefaultAsync();
```

🔥 Why This Is Better:

- **Faster:** Loads only necessary data.
- **Less Memory:** Doesn't bring extra columns you don't need.
- **More Efficient:** Makes your app run smoother.

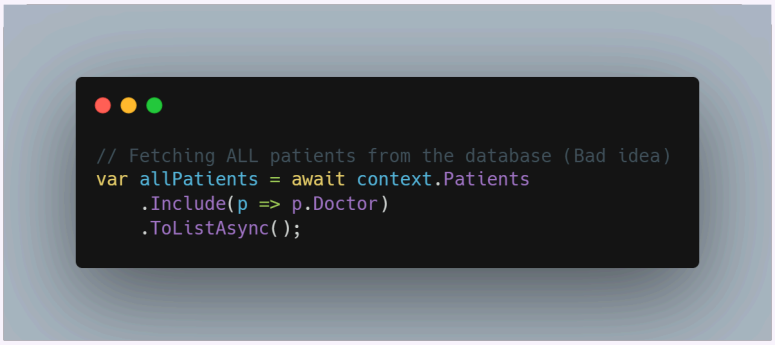
Always use `.Select()` to pick only what you need! 🚀

🗣️ 3. Over fetching Data

When you **retrieve too much data at once**, your app can **become slow and use too much memory**. This happens when you load entire tables or big datasets without limiting how much you get.

❌ The Wrong Way (Fetching Everything)

If you have a `Patients` table with thousands of records and you load them all at once:



```
// Fetching ALL patients from the database (Bad idea)
var allPatients = await context.Patients
    .Include(p => p.Doctor)
    .ToListAsync();
```

This **loads everything** into memory, which can be very slow and heavy.

✅ The Right Way (Using Pagination)

Instead, **load data in small parts** (pages) using `.Skip()` and `.Take()`.

```
int pageSize = 50;     // Number of records per page
int pageNumber = 1;    // Page number to load

var patients = await context.Patients
    .AsNoTracking()
    .OrderBy(p => p.Name)
    .Skip((pageNumber - 1) * pageSize)
    .Take(pageSize)
    .ToListAsync();
```

This loads **only 50 patients at a time**. Much faster and uses less memory!

🔥 Cursor-Based Pagination (Another Way)

Instead of using page numbers, you can **load data based on the last loaded record's ID**. This is more efficient for huge tables.

```
int pageSize = 50;
int lastId = 102; // Last loaded patient's ID from previous query

var patients = await context.Patients
    .AsNoTracking()
    .OrderBy(p => p.Id)
    .Where(p => p.Id > lastId)
    .Take(pageSize)
    .ToListAsync();
```

📌 Which One To Use?

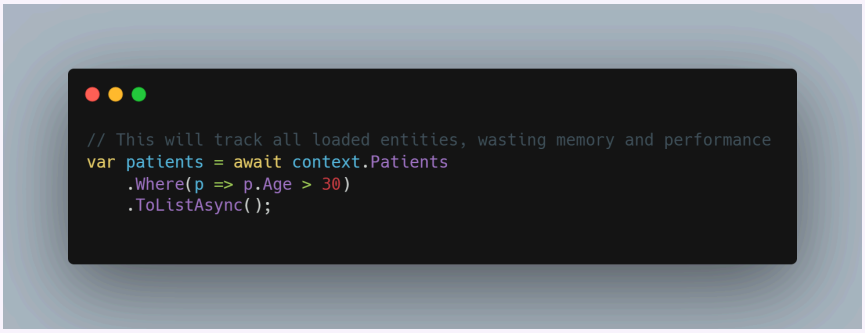
- ✅ **Use Offset-Based Pagination (`Skip` and `Take`)** when you want to access **any specific page** by number.
- ✅ **Use Cursor-Based Pagination** when you **only need next or previous pages**. It's faster for large tables.

🗣️ 4. Not Using `AsNoTracking`

By default, EF Core **tracks all entities** you fetch. This tracking is used to detect changes when you update data, but it's **unnecessary for read-only queries** and **slows things down**.

❌ The Wrong Way (Tracking Everything)

If you only want to **view data**, but you fetch it like this:



```
// This will track all loaded entities, wasting memory and performance
var patients = await context.Patients
    .Where(p => p.Age > 30)
    .ToListAsync();
```

EF Core is tracking every patient it loads, even though you're not planning to **edit or save** them.

✅ The Right Way (Using `AsNoTracking()`)

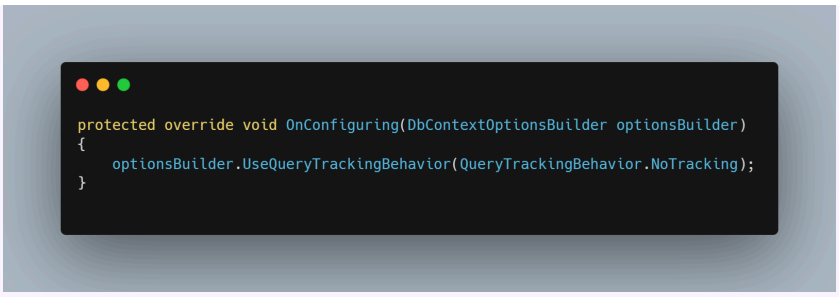
For read-only queries, use `.AsNoTracking()` to **tell EF Core not to track the data**.

```
// Using AsNoTracking() to improve performance
var patients = await context.Patients
    .AsNoTracking()
    .Where(p => p.Age > 30)
    .ToListAsync();
```

This **boosts performance** and **reduces memory usage**.

🔥 Make All Queries Non-Trackable (If Needed)

If your `DbContext` is **only used for reading data**, you can make all queries non-tracking by default.



```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
}
```

Now, every query is treated as `AsNoTracking()` unless you **explicitly enable tracking**.

📌 When To Use `AsNoTracking()` :

- ✅ When you're **just reading data** and don't need to update it.
- ✅ When **performance matters** (like displaying lists, reports, etc.).

🗣️ 5. Using Eager Loading Unwisely

Eager Loading is when you **load related data along with the main data** using `.Include()` and `.ThenInclude()`. It's helpful when you need related data immediately, but **if you load too much, it can slow things down**.

❌ The Wrong Way (Loading Too Much Data)

Imagine you have `Patients` who each have a `Doctor`, and each `Doctor` has a `Department`. If you fetch **all patients with their doctors and departments** like this:

```
// Fetching all patients, their doctors, and departments - Can be too much data!
var patients = await context.Patients
    .AsNoTracking()
    .Include(p => p.Doctor)
        .ThenInclude(d => d.Department)
    .ToListAsync();
```

This **loads everything in a single query**. If you have thousands of patients, doctors, and departments, this query will be **very slow**.

✅ The Right Way (Limiting Data with Filters)

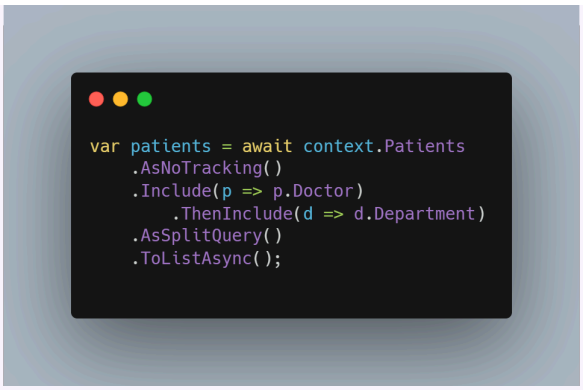
Instead of loading all related data, you can **filter what you load**. For example, if you only want **patients assigned to a specific department**:

```
// Loading only patients from a specific department
var patients = await context.Patients
    .Include(p => p.Doctor)
        .ThenInclude(d => d.Department)
    .Where(p => p.Doctor.Department.Name == "Ophthalmology")
    .ToListAsync();
```

This is **much faster** because you're not loading unrelated data.

🔥 Using `AsSplitQuery()` for Better Performance

If you have a **complex query** with multiple `.Include()`s, you can split it into **separate queries** using `.AsSplitQuery()`.



```
var patients = await context.Patients
    .AsNoTracking()
    .Include(p => p.Doctor)
        .ThenInclude(d => d.Department)
    .AsSplitQuery()
    .ToListAsync();
```

This generates **separate queries** for `Patients`, `Doctors`, and `Departments`. It's usually **faster and more efficient** for large datasets.

Aspect	`AsSplitQuery`	Eager Loading
Best For	Complex `JOINs`, deep relationships	Simple or flat relationships
Number of Queries	Multiple queries (more round trips)	Single query (less round trips)
Network Traffic	Higher if too many queries	Lower, but heavy if too much data is loaded
Consistency	Risk of inconsistency	Consistent data retrieval
Performance	Better for complex graphs	Better for simple queries
Risk of N+1 Problem	High if misused	Can occur if not configured properly

Summary

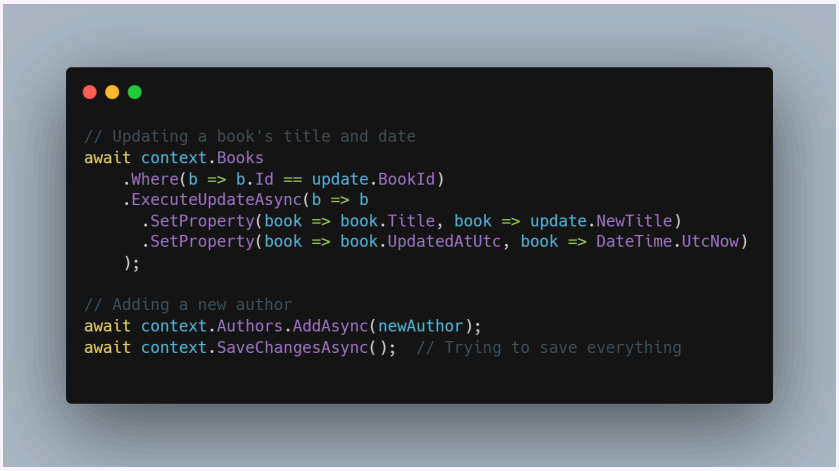
- ✅ Use `AsSplitQuery` if you see **too many complex JOINs** in the SQL query.
- ✅ Use Eager Loading if you want to **load related data all at once** with a simple query.

🗣️ 6. Ignoring Transactions When Using Batch Operations

When you update or delete multiple records at once using `ExecuteUpdateAsync` or `ExecuteDeleteAsync`, **EF Core does not track these changes**. This means that if something goes wrong **afterwards**, like adding a new record that fails, the previous updates or deletes **won't be automatically undone**.

❌ The Wrong Way (No Transaction Handling)

Let's say you have a **Books** table and an **Authors** table. You're trying to **update some books** and then **add a new author**.



```
// Updating a book's title and date
await context.Books
    .Where(b => b.Id == update.BookId)
    .ExecuteUpdateAsync(b => b
      .SetProperty(book => book.Title, book => update.NewTitle)
      .SetProperty(book => book.UpdatedAtUtc, book => DateTime.UtcNow)
    );

// Adding a new author
await context.Authors.AddAsync(newAuthor);
await context.SaveChangesAsync();  // Trying to save everything
```

🔥 The Problem:

If adding the new author fails for any reason, the update to the book **already happened**. It **won't be rolled back automatically**, and now your data is inconsistent.

✅ The Right Way (Using Transactions)

Wrapping the whole operation in a **transaction** ensures that **either everything is saved successfully or nothing is saved at all**.

```
using var transaction = await _context.Database.BeginTransactionAsync();
try
{
    // Update the book
    await context.Books
        .Where(b => b.Id == update.BookId)
        .ExecuteUpdateAsync(b => b
            .SetProperty(book => book.Title, book => update.NewTitle)
            .SetProperty(book => book.UpdatedAtUtc, book => DateTime.UtcNow)
        );

    // Add the new author
    await context.Authors.AddAsync(newAuthor);
    await context.SaveChangesAsync();

    // Commit the transaction if everything is successful
    await transaction.CommitAsync();
}
catch (Exception)
{
    // Rollback if something goes wrong
    await transaction.RollbackAsync();
    throw;
}
```

📌 Why This Works:

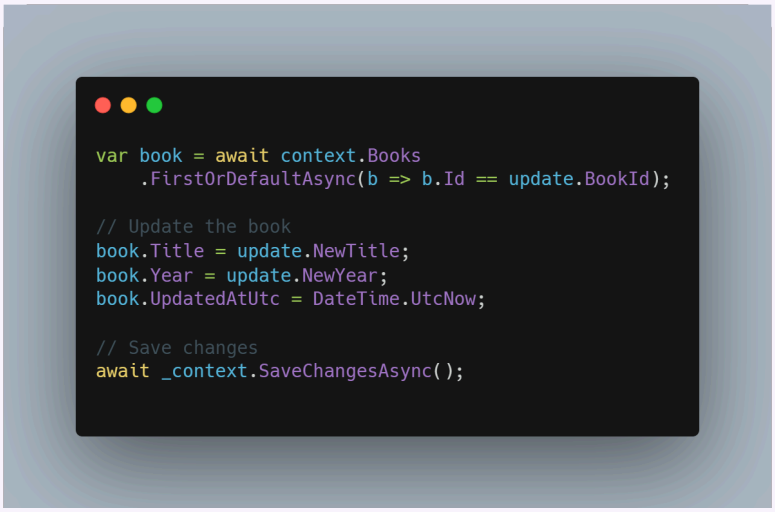
- ✅ **Atomic Operations:** If something fails, nothing gets saved.
- ✅ **Data Integrity:** Everything succeeds or everything fails. No half-done updates.
- ✅ **Safety:** Helps prevent data corruption or inconsistency.

🗣️ 7. Ignoring Concurrency Handling

When multiple users try to **edit the same record** at the same time, **one user's changes can overwrite the other's** without any warning. This is called a **Concurrency Conflict**.

❌ The Wrong Way (No Concurrency Handling)

If two users try to **update the same book**, only the last change will be saved, and the first one will be lost.



```
var book = await context.Books
    .FirstOrDefaultAsync(b => b.Id == update.BookId);

// Update the book
book.Title = update.NewTitle;
book.Year = update.NewYear;
book.UpdatedAtUtc = DateTime.UtcNow;

// Save changes
await _context.SaveChangesAsync();
```

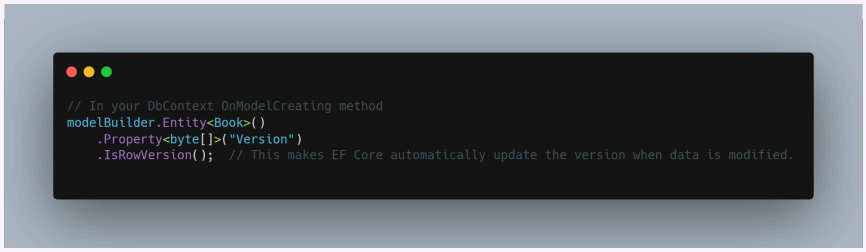
🔥 The Problem:

- **No protection** against multiple updates happening at the same time.
- Changes made by the first user are **overwritten by the second user** without any warning.

✅ The Right Way (Using Concurrency Tokens)

Adding a **RowVersion column** to the entity allows EF Core to detect if the data has been changed by someone else before saving.

📌 Step 1: Configure the Entity



```
// In your DbContext OnModelCreating method
modelBuilder.Entity<Book>()
    .Property<byte[]>("Version")
    .IsRowVersion();  // This makes EF Core automatically update the version when data is modified.
```

📌 Step 2: Handling Concurrency Conflicts

```
var book = await context.Books
    .FirstOrDefaultAsync(b => b.Id == update.BookId);

book.Title = update.NewTitle;
book.Year = update.NewYear;
book.UpdatedAtUtc = DateTime.UtcNow;

try
{
    await _context.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException ex)
{
    // Handle the concurrency conflict

    // Option 1: Reload from database to get the current data
    var entry = ex.Entries.Single();
    entry.Reload();  // This will overwrite the local changes with the database values.

    // Option 2: Merge changes manually
    var databaseValues = entry.GetDatabaseValues();
    var currentValues = entry.CurrentValues;

    // Example: Prefer the current title but accept the rest from the database
    entry.OriginalValues.SetValues(databaseValues);
    entry.CurrentValues["Title"] = currentValues["Title"];

    // Save again after resolving the conflict
    await _context.SaveChangesAsync();
}
```

📌 Why This Works:

- ✅ **Detection of Conflicts:** EF Core detects changes by comparing the `Version` value.
- ✅ **Handling Options:** You can decide to **merge changes**, **overwrite them**, or **reject the update**.
- ✅ **Safety:** Prevents users from unintentionally overwriting each other's updates.

🎯 Concurrency Handling Example - Step by Step

📌 Scenario:

- You have a `Book` record with `Id = 1` and initial `Version = 1`.
- Two users `User1` and `User2` try to **update the same book**.

✅ 1. Initial Data in Database

Id	Title	Year	Version
1	Old Title	2000	1

👤 2. User1 Fetches the Book

```
var book = await context.Books.FirstOrDefaultAsync(b => b.Id == 1);
```

- 📌 **Version = 1** (in memory).

👤 3. User2 Fetches the Same Book

```
var book = await context.Books.FirstOrDefaultAsync(b => b.Id == 1);
```

- 📌 **Version = 1** (in memory).

✍️ 4. User1 Makes Changes and Saves

```
book.Title = "New Title by User1";
await context.SaveChangesAsync();
```

- **Database Updated:**
  - `Title = "New Title by User1"`
  - `Version = 2` (automatically incremented by the database)

Id	Title	Year	Version
1	New Title by User1	2000	2

✍️ 5. User2 Makes Changes and Tries to Save

```
book.Title = "New Title by User2";
await context.SaveChangesAsync();
```

- **Error Thrown:** `DbUpdateConcurrencyException` ❌
- Why? Because User2's `Version = 1` but the database has `Version = 2`.

⚠️ 6. Handling the Conflict (In User2's Code)

```
try
{
    await context.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException ex)
{
    var entry = ex.Entries.Single();

    var databaseValues = entry.GetDatabaseValues(); // Latest data from the database (User1's changes)
    var currentValues = entry.CurrentValues; // User2's unsaved changes

    // Example: Keep User2's title change but update the version to match the database.
    entry.OriginalValues.SetValues(databaseValues);
    entry.CurrentValues["Title"] = currentValues["Title"]; // Keeping User2's title change

    await context.SaveChangesAsync(); // Saving successfully with updated version
}
```

✅ 7. Final Data in Database

Id	Title	Year	Version
1	New Title by User2	2000	3

📌 Summary:

- **User1's changes were saved first.**
- **User2 got an error because the version was outdated.**
- **User2's changes were then applied successfully after handling the conflict.**

🗣️ 8. Not Using Migrations

EF Core Migrations help you keep your **database schema** in sync with your **application's data model**. If you modify your entities without updating the database through migrations, your application will eventually crash or behave unexpectedly.

❌ The Wrong Way (Manual Schema Modification)

- Modifying the **database directly** using SQL commands or a database management tool.
- Skipping migrations and expecting EF Core to recognize the changes automatically.

❌ **Problem:**

- The database is out of sync with your application model.
- You will encounter **errors when retrieving or updating data**.
- Hard to track changes or revert problematic changes.

✅ The Right Way (Using EF Core Migrations)

Instead of editing the database manually, use migrations to safely and consistently apply changes.

📌 Step 1: Add Migration

Generate a new migration file with changes detected by EF Core.

```
Add-Migration AddNewPropertyToMessage
```

- This command will create a new migration file under the `Migrations` folder.
- The file will contain the SQL commands needed to update the database schema.

📌 Step 2: Update Database

Apply the migration to your database.

```
Update-Database
```

- This command will run the generated SQL against your database to apply the changes.

🗣️ 9. Forgetting to Dispose Manually Created DbContext

When you manually create a `DbContext` using `IDbContextFactory`, you need to **dispose of it properly** to avoid memory leaks and resource exhaustion.

❌ The Wrong Way (Not Disposing `DbContext`)

```
private readonly IDbContextFactory<ApplicationDbContext> _contextFactory;

// 🚫 BAD CODE: Memory Leak, DbContext is not disposed
public async Task StartAsync(CancellationToken cancellationToken)
{
    var context = await _contextFactory.CreateDbContextAsync();
    var books = await context.Books.ToListAsync();
}
```

⚠️ Why This Is Bad:

- When `DbContext` is not disposed, **connections to the database** remain open, consuming memory and resources.
- Memory leaks can cause your application to slow down or even crash.

✅ The Right Way (Using `using` Statement to Dispose `DbContext`)

```
private readonly IDbContextFactory<ApplicationDbContext> _contextFactory;

// ✅ Correctly disposing of DbContext
public async Task StartAsync(CancellationToken cancellationToken)
{
    using var context = await _contextFactory.CreateDbContextAsync();
    var books = await context.Books.ToListAsync();
}
```

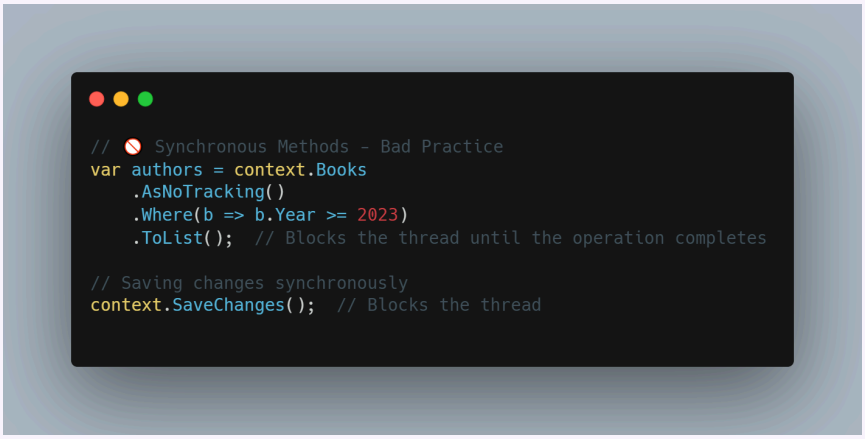
📌 Explanation:

- The `using` statement ensures that the `DbContext` is disposed **as soon as it goes out of scope**.
- This releases the database connection and other resources used by the `DbContext`.

🗣️ 10. Not Using Asynchronous Methods

When you **don't use asynchronous methods** in EF Core, your application can become **slow and unresponsive**, especially when dealing with **large datasets or multiple concurrent requests**.

❌ The Wrong Way (Using Synchronous Methods)



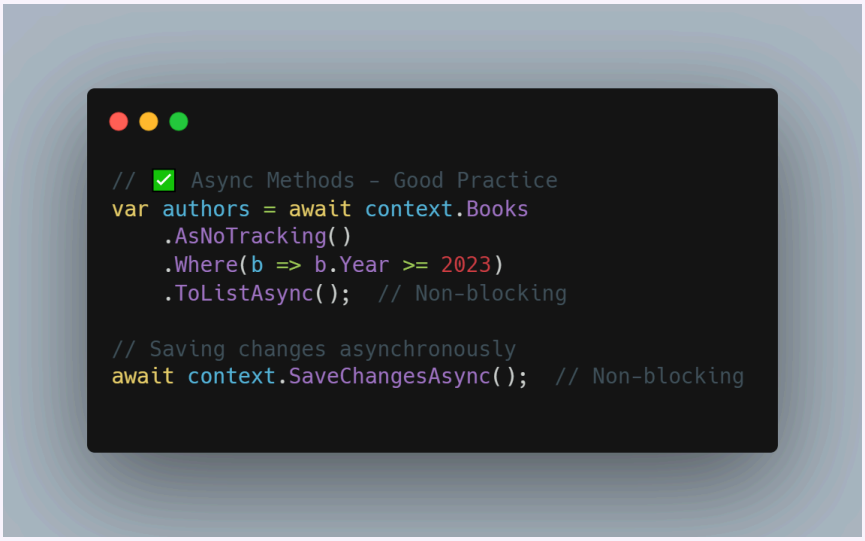
```
// 🚫 Synchronous Methods - Bad Practice
var authors = context.Books
    .AsNoTracking()
    .Where(b => b.Year >= 2023)
    .ToList();  // Blocks the thread until the operation completes

// Saving changes synchronously
context.SaveChanges();  // Blocks the thread
```

⚠️ Why This Is Bad:

1. **Blocks Threads:** Operations like `.ToList()` and `.SaveChanges()` block the thread until the operation completes, causing delays.
2. **Reduced Scalability:** If multiple requests are handled simultaneously, blocking threads will limit performance.
3. **Poor User Experience:** Users may experience slow response times or frozen UI.

✅ The Right Way (Using Asynchronous Methods)



```
// ✅ Async Methods - Good Practice
var authors = await context.Books
    .AsNoTracking()
    .Where(b => b.Year >= 2023)
    .ToListAsync();  // Non-blocking

// Saving changes asynchronously
await context.SaveChangesAsync();  // Non-blocking
```

📌 Explanation:

- `ToListAsync()` and `SaveChangesAsync()` are non-blocking operations.
- They **free up the thread** to handle other requests or tasks while waiting for the database operation to complete.
- This approach makes your application **more responsive and scalable**.

📌 Important Tips:

1. ✅ **Always Use Async Methods:** Use async versions of EF Core methods whenever possible (`ToListAsync()`, `FindAsync()`, `SaveChangesAsync()`, etc.).
2. ✅ **Use `await` Properly:** Ensure you are using `await` when calling async methods, otherwise it will still block the thread.
3. ✅ **Consistent Async Usage:** Make sure the entire call chain is asynchronous (e.g., Controllers, Services, Repositories). Mixing sync and async can lead to deadlocks or performance issues.

🗣️ 11 . Unknow How to use Use Compiled Queries

📌 What are Compiled Queries in EF Core?

When you write a **LINQ query** in EF Core, two main steps happen:

1. **Translating your C# code to a SQL query.**
2. **Executing that SQL query on the database.**

The problem is, every time you run the **same query** with different parameters (e.g., retrieving records by different `Id`s), EF Core **translates the query from scratch** every single time. This translation process takes time and consumes resources.

✅ The Solution: Compiled Queries

Instead of making EF Core translate the query every time, it **caches the SQL query** in memory and reuses it directly. This can **significantly improve performance**, especially if you are executing the same query repeatedly.

🎯 Practical Example (I'll explain both Synchronous and Asynchronous versions)

🔍 1. Regular Query without Compiled Queries

```
var book = context.Books
    .Include(b => b.Author)
    .FirstOrDefault(b => b.Title == "Clean Code");
```

If you run this query **ten times**, EF Core **translates it to SQL** every single time, which is inefficient.

🔥 2. The Same Query Using Compiled Queries (Synchronous Version)

```
private static readonly Func<ApplicationDbContext, string, Book?> BookByTitle
    = EF.CompileQuery((ApplicationDbContext context, string title) =>
        context.Books
            .Include(b => b.Author)
            .FirstOrDefault(b => b.Title == title)
    );

public Book? GetBookByTitle(string title)
    => BookByTitle(this, title);
```

💡 What's Happening Here?

- `BookByTitle` is a variable that **stores the query** you wrote, but it's **already translated to SQL** and **cached** in memory.
- When you call `GetBookByTitle("Clean Code")`, **EF Core doesn't translate it again**—it just executes it immediately.
- Performance will be **much faster** if you run this query repeatedly.

🔥 3. The Same Query Using Compiled Queries (Asynchronous Version)

```
private static readonly Func<ApplicationDbContext, string, CancellationToken, Task<Book?>>
BookByTitleAsync
    = EF.CompileAsyncQuery((ApplicationDbContext context, string title, CancellationToken
cancellationToken) =>
        context.Books
            .Include(b => b.Author)
            .FirstOrDefault(b => b.Title == title)
    );

public async Task<Book?> GetBookByTitleAsync(string title, CancellationToken cancellationToken)
    => await BookByTitleAsync(this, title, cancellationToken);
```

💡 What's Different Here?

- Same concept, but using `CompileAsyncQuery` for **asynchronous execution**.
- Accepts a `CancellationToken` in case you need to **cancel the operation midway** if it's taking too long (important for performance and responsiveness in web apps).

✅ Do I have to send the same Title every time to benefit from the Compiled Query?

No, you can send different titles each time and still benefit from the Compiled Query. ✅

📌 When to Use Compiled Queries?

1. **When you execute the same query frequently with different parameters.**
2. **When you have a lot of read operations and want to boost performance.**
3. **When you notice your application is slow due to repeated query translation.**

🗣️ 12 -unknow when to use Raw SQL Queries

✅ Using Raw SQL Queries in EF Core (Simply Explained)

🔍 Why Use Raw SQL Queries?

Sometimes EF Core can't handle very complex queries efficiently. Using raw SQL can:

- **Skip EF Core's translation process** (faster).
- Use database-specific features like `FOR UPDATE` locks.
- Write complicated queries directly instead of trying to fit them into LINQ.

📌 How to Use Raw SQL Queries

1. Basic Raw SQL Query (Simple Fetching)

```
var year = 2020;

var newBooks = await context.Database
    .SqlQuery<Book>($"SELECT * FROM Books WHERE Year > {year}")
    .ToListAsync();
```

✅ This works but is not safe against SQL injection. Avoid using this method with user inputs.

2. Safe Raw SQL Query (Using Parameters)

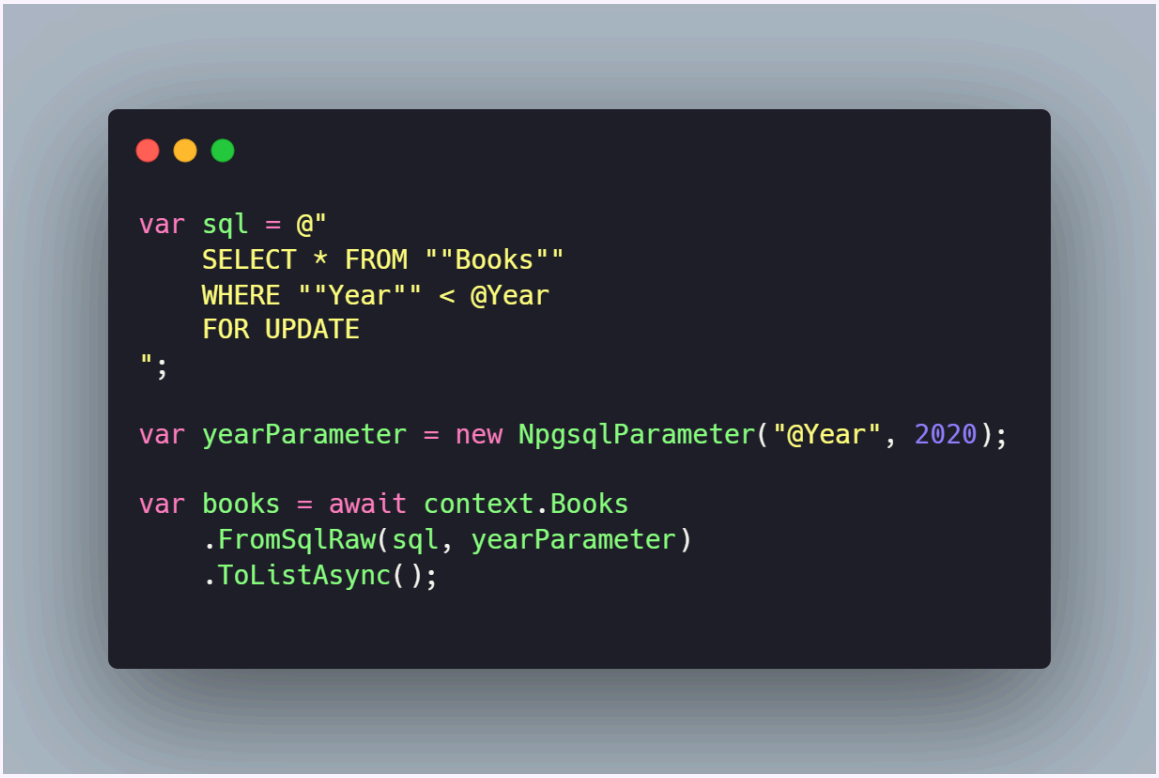
```
var year = 2020;

var sql = "SELECT * FROM Books WHERE Year > @year";
var yearParameter = new SqlParameter("@year", year);

var newBooks = await context.Books
    .FromSqlRaw(sql, yearParameter)
    .ToListAsync();
```

✅ **Always use parameters** like this to prevent SQL injection. This is the preferred way.

3. Using Raw SQL for Advanced Operations (e.g., Locks)



```
var sql = @"
    SELECT * FROM ""Books""
    WHERE ""Year"" < @Year
    FOR UPDATE
";

var yearParameter = new NpgsqlParameter("@Year", 2020);

var books = await context.Books
    .FromSqlRaw(sql, yearParameter)
    .ToListAsync();
```

✅ You can use special SQL commands like `FOR UPDATE` which EF Core doesn't support directly.

🗣️ 13- Un Introduce Caching

🔍 Why Use Caching?

Caching **improves performance and reduces database load** by storing frequently accessed data in memory or distributed caches like Redis. It helps avoid repeated database queries, especially for **read-heavy operations**.

📌 Implementing Caching in .NET

1. In-Memory Caching (Simple Approach)

You can use `IMemoryCache` for caching data within the same application instance.

```
using Microsoft.Extensions.Caching.Memory;

private readonly IMemoryCache _cache;
private readonly ApplicationDbContext _context;

public YourService(IMemoryCache cache, ApplicationDbContext context)
{
    _cache = cache;
    _context = context;
}

public async Task<Book?> GetBookAsync(Guid id, CancellationToken cancellationToken)
{
    var cacheKey = $"Book_{id}";

    if (_cache.TryGetValue(cacheKey, out Book? book))
    {
        return book; // Return the cached item if it exists.
    }

    book = await _context.Books
        .Include(b => b.Author)
        .FirstOrDefaultAsync(b => b.Id == id, cancellationToken);

    if (book != null)
    {
        _cache.Set(cacheKey, book, TimeSpan.FromMinutes(5)); // Store the book in cache for 5 minutes.
    }

    return book;
}
```

2. Distributed Caching (Redis Example)

For **distributed environments** or when you want to **share cache data across multiple instances**, use Redis.

Configure Redis in `Program.cs`:

```
builder.Services.AddStackExchangeRedisCache(options ⇒
{
   options.Configuration = "localhost:6379";
});
```

Usage Example:

```
using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;

private readonly IDistributedCache _cache;
private readonly ApplicationDbContext _context;

public YourService(IDistributedCache cache, ApplicationDbContext context)
{
    _cache = cache;
    _context = context;
}

public async Task<Book?> GetBookAsync(Guid id, CancellationToken cancellationToken)
{
    var cacheKey = $"Book_{id}";
    var cachedBook = await _cache.GetStringAsync(cacheKey, cancellationToken);

    if (!string.IsNullOrEmpty(cachedBook))
    {
        return JsonSerializer.Deserialize<Book>(cachedBook);
    }

    var book = await _context.Books
        .Include(b => b.Author)
        .FirstOrDefaultAsync(b => b.Id == id, cancellationToken);

    if (book != null)
    {
        var serializedBook = JsonSerializer.Serialize(book);
        await _cache.SetStringAsync(cacheKey, serializedBook, new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5)
        }, cancellationToken);
    }

    return book;
}
```

3. Using HybridCache (Recommended in .NET 9+)

.NET 9 introduces `HybridCache`, which combines memory and distributed caching for **scalability and performance**.

It prevents **Cache Stampede** by using techniques like **locking and request coalescing**.

✅ Bonus Mistakes to Avoid

1. Saving Changes in the Loop

Mistake: Calling `SaveChanges()` inside a loop will **send a database request for each iteration**, which is extremely inefficient.

❌ **The Wrong Way**

```
foreach (var book in books)
{
    book.UpdatedAtUtc = DateTime.UtcNow;
    context.Update(book);
    await context.SaveChangesAsync(); // Bad: Executed for each item in the loop
}
```

✅ **The Right Way**

```
foreach (var book in books)
{
    book.UpdatedAtUtc = DateTime.UtcNow;
    context.Update(book);
}
await context.SaveChangesAsync(); // Good: Only one call after all updates are made
```

💡 **Tip:** You can also use `ExecuteUpdateAsync()` if you don't need to track entities.

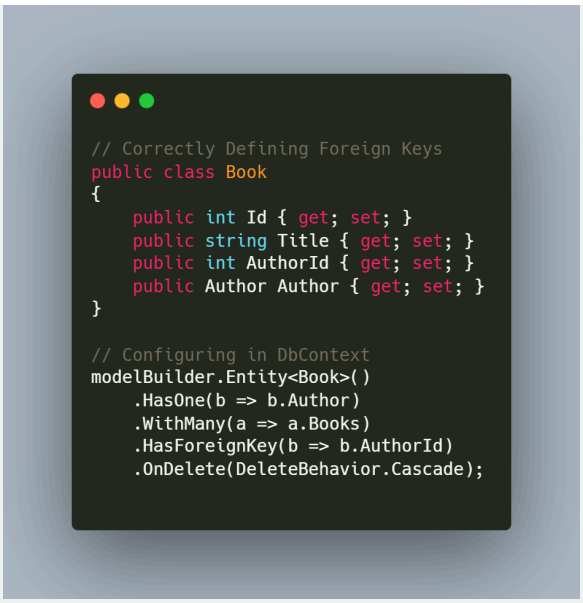
2. Not Configuring Relationships Properly

Mistake: If you **don't define relationships correctly** (e.g., missing foreign keys, incorrect `DeleteBehavior`, etc.), it can cause **data inconsistencies, errors, and performance issues**.

❌ **The Wrong Way**

```
// No Foreign Key Definition
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public Author Author { get; set; } // Relationship not properly configured
}
```

✅ **The Right Way**



```
// Correctly Defining Foreign Keys
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Author Author { get; set; }
}

// Configuring in DbContext
modelBuilder.Entity<Book>()
    .HasOne(b => b.Author)
    .WithMany(a => a.Books)
    .HasForeignKey(b => b.AuthorId)
    .OnDelete(DeleteBehavior.Cascade);
```

💡 **Tip:** Always **explicitly configure relationships** to avoid unexpected behavior.

3. Ignoring Database Normalization

Mistake: Poorly designed schemas can cause **data redundancy, anomalies, and performance issues**. For example, storing a patient's department name directly in the `Patient` table instead of linking it through a foreign key.

❌ **The Wrong Way**

```
// Not Normalized - Department name stored directly
public class Patient
{
   public int Id { get; set; }
   public string Name { get; set; }
   public string DepartmentName { get; set; } // Data redundancy
}
```

✅ **The Right Way**

```
// Normalized - Using Foreign Keys
public class Patient
{
   public int Id { get; set; }
   public string Name { get; set; }
   public int DepartmentId { get; set; }
   public Department Department { get; set; }
}
```

💡 **Tip:** Normalize your database to **reduce redundancy** and **improve consistency**.

4. Ignoring Connection Pooling

Mistake: Not utilizing connection pooling in **high database access scenarios** can lead to performance degradation and even cause a "Max Connection Reached" error.

✅ **Solution**

- EF Core uses connection pooling by default when using a connection string with `Microsoft.Data.SqlClient` or `Npgsql`.
- You can configure pooling with your connection string:

```
"Server=myServer;Database=myDB;User Id=myUser;Password=myPass;Pooling=true;Max Pool Size=100;"
```

💡 **Tips:**

- Adjust `Max Pool Size` based on your application's load.
- Monitor the pool usage to optimize performance.
- Use `DbContextFactory` if you need to create many `DbContext` instances.

How To Create Migrations For Multiple Databases in EF Core

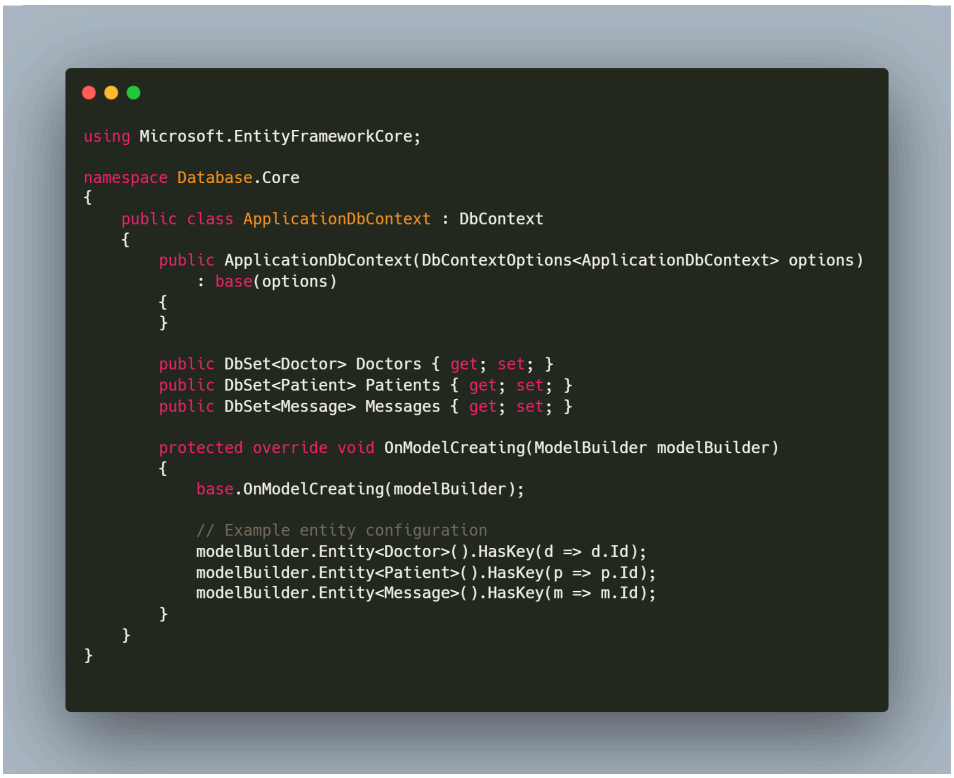
If you want to **support multiple databases like (Postgres, SQL Server, SQLite.)** within a **single web application (`MyHospitaloo`)**, and keep **separate migrations** for each database provider, follow these steps:

🌳 Project Structure

1. **MyHospitaloo (WebApp)** - Contains your `ApplicationDbContext`, migrations, and all configurations.
2. **Database.Core** - Contains your entities and `DbContext`.
3. create three Classes Library for Every Migration ⇒(one for every **Postgresand one for SQL Server, and one for SQLite.)**

📦 Database.Core (Shared Library)

📂 `ApplicationDbContext.cs`



```
using Microsoft.EntityFrameworkCore;

namespace Database.Core
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        public DbSet<Doctor> Doctors { get; set; }
        public DbSet<Patient> Patients { get; set; }
        public DbSet<Message> Messages { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            // Example entity configuration
            modelBuilder.Entity<Doctor>().HasKey(d => d.Id);
            modelBuilder.Entity<Patient>().HasKey(p => p.Id);
            modelBuilder.Entity<Message>().HasKey(m => m.Id);
        }
    }
}
```

📦 MyHospitaloo (WebApp)

📂 Program.cs

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

var provider = builder.Configuration.GetValue<string>("Provider");

if (provider == "Postgres")
{
    builder.Services.AddDbContext<ApplicationDbContext>(options =>
        options.UseNpgsql(builder.Configuration.GetConnectionString("Postgres"))
               .MigrationsAssembly("MyHospitaloo.PostgresMigrations"));
}
else if (provider == "SqlServer")
{
    builder.Services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(builder.Configuration.GetConnectionString("SqlServer"))
               .MigrationsAssembly("MyHospitaloo.SqlServerMigrations"));
}
else if (provider == "SQLite")
{
    builder.Services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlite(builder.Configuration.GetConnectionString("SQLite"))
               .MigrationsAssembly("MyHospitaloo.SqliteMigrations"));
}

var app = builder.Build();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.MapRazorPages();
app.Run();
```

📂 appsettings.json

```
{
  "ConnectionStrings": {
    "Postgres": "Host=localhost;Database=MyHospitaloo_Postgres;Username=postgres;Password=yourpassword",
    "SqlServer": "Server=localhost;Database=MyHospitaloo_SqlServer;User Id=sa;Password=yourpassword;",
    "SQLite": "Data Source=MyHospitaloo_Sqlite.db"
  },
  "Provider": "Postgres"
}
```

📦 Creating Marker Interfaces for Migrations

To easily identify which migration assembly to use without hardcoding, create a **marker interface** for each migration project.

📂 MyHospitaloo.PostgresMigrations

```
namespace MyHospitaloo.PostgresMigrations
{
    public interface IPostgresMigrationMarker {}
}
```

📂 MyHospitaloo.SqlServerMigrations

```
namespace MyHospitaloo.SqlServerMigrations
{
    public interface ISqlServerMigrationMarker {}
}
```

📂 MyHospitaloo.SqliteMigrations

```
namespace MyHospitaloo.SqliteMigrations
{
    public interface ISqliteMigrationMarker {}
}
```

🔨 Creating Migrations For Each Database

Each migration will have its own folder under its respective migration project.

📌 For Postgres

```
dotnet ef migrations add InitialPostgresMigration --context ApplicationDbContext --output-dir Migrations --project MyHospitaloo.PostgresMigrations
```

📌 For SQL Server

```
dotnet ef migrations add InitialSqlServerMigration --context ApplicationDbContext --output-dir Migrations --project MyHospitaloo.SqlServerMigrations
```

📌 For SQLite

```
dotnet ef migrations add InitialSqliteMigration --context ApplicationDbContext --output-dir Migrations --project MyHospitaloo.SqliteMigrations
```

📦 Applying Migrations (Database Update)

To apply migrations for each provider, use:

📌 For Postgres

```
dotnet ef database update --context ApplicationDbContext --project MyHospitaloo.PostgresMigrations
```

📌 For SQL Server

```
dotnet ef database update --context ApplicationDbContext --project MyHospitaloo.SqlServerMigrations
```

📌 For SQLite

```
dotnet ef database update --context ApplicationDbContext --project MyHospitaloo.SqliteMigrations
```

💡 Explanation

1. 🔀 **Single Web Application Supporting Multiple Databases:**

   All your database logic is in one `DbContext` (`ApplicationDbContext`), but you can switch between databases using a configuration setting (`Provider`).

2. 📂 **Separate Migration Projects:**

   By creating separate projects (`MyHospitaloo.PostgresMigrations`, `MyHospitaloo.SqlServerMigrations`, `MyHospitaloo.SqliteMigrations`), you ensure that migrations are isolated.

3. 🔧 **Marker Interfaces:**

   Adding empty marker interfaces helps you reference migration assemblies dynamically without hardcoding their names in the `Program.cs`.

4. 🔒 **Independent Configuration:**

   You can easily switch between different providers by changing the `Provider` key in your `appsettings.json`.

🥳

We Finished , With My Best Wishes ,Wait For the Next

Connection with me to see all new : https://www.linkedin.com/in/ahmed-hany-899a9a321?utm_source=share&utm_campaign=share_via&utm_content=profile&utm_med