WEEK –3 HANDSON

Lab 1: Understanding ORM with a Retail Inventory System

1. What is ORM?

• Explain how ORM maps C# classes to database tables.

**ORM** (Object-Relational Mapping) is a technique that lets you interact with a relational database (like SQL Server) using **object-oriented programming (OOP)** principles.

In the context of C# and EF Core:

* A **class** (e.g., Product) maps to a **table** in the database (Products).
* **Class properties** (e.g., Name, Price) map to **columns** in the table.
* **Relationships** (e.g., one-to-many, many-to-many) are defined using navigation properties and foreign keys.

• Benefits: Productivity, maintainability, and abstraction from SQL.

* **Productivity**: No need to write raw SQL queries — you use LINQ and C#.
* ️ **Maintainability**: Changes in the model reflect easily in the database.
* **Abstraction**: Developers can focus on business logic without knowing SQL.

2. EF Core vs EF Framework:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Entity Framework Core (EF Core)** | **Entity Framework (EF 6)** |
| Platform | Cross-platform (.NET Core, .NET 6/7/8) | Windows-only (.NET Framework) |
| Performance | Faster, more lightweight | Slower, older architecture |
| Features | LINQ, async queries, compiled models, shadow properties, global filters | Basic LINQ and tracking, no compiled models |
| Flexibility | Modular, good for microservices, cloud apps | Monolithic, older web forms/WPF apps |
| Latest Version | **EF Core 8.0 (Latest)** | EF 6 is no longer actively developed |

3. EF Core 8.0 Features

EF Core 8.0 introduced **modern enhancements** that boost both flexibility and performance:

### **JSON Column Mapping**

* Store and query JSON data directly in database columns.
* Example: public Dictionary<string, string> Specifications { get; set; } can be saved as JSON.

### **Compiled Models**

* Improves app startup time and runtime performance by compiling the EF model at build time.
* Great for large-scale apps.

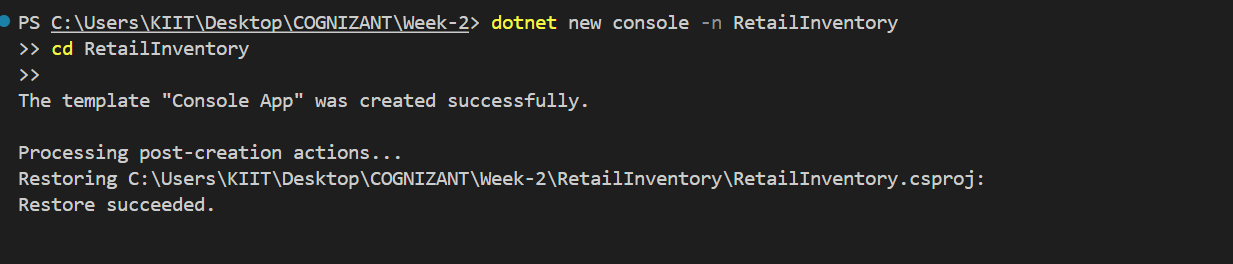
### **Interceptors**

* Hook into EF Core’s operations (e.g., logging, auditing, soft delete).
* Lets you intercept and customize database queries and commands.

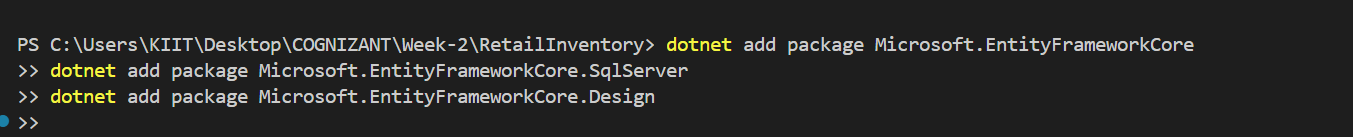
### **Bulk Updates & Deletes**

* Native support for ExecuteUpdate and ExecuteDelete, which perform **efficient, direct SQL** without loading entities into memory.

4. Create a .NET Console App:



5. Install EF Core Packages:



Lab 2: Setting Up the Database Context for a Retail Store

1. Create Models:

using System.Collections.Generic;

namespace RetailInventory.Models;

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

public List<Product> Products { get; set; } = new();

}

namespace RetailInventory.Models;

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int CategoryId { get; set; }

public Category Category { get; set; }

}

2. Create AppDbContext:

using Microsoft.EntityFrameworkCore;

using RetailInventory.Models; // ✅ Required!

public class AppDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

public DbSet<Category> Categories { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

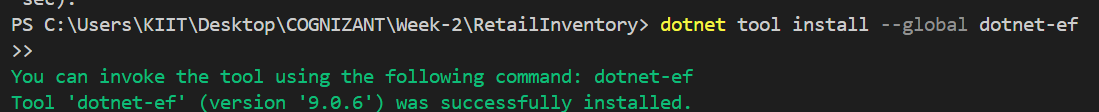
optionsBuilder.UseSqlServer(@"Server=bt-2205576\SQLEXPRESS;Database=RetailDB;Trusted\_Connection=True;TrustServerCertificate=True;");

}

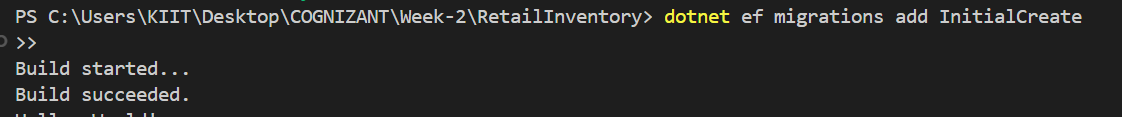
}

Lab 3: Using EF Core CLI to Create and Apply Migrations

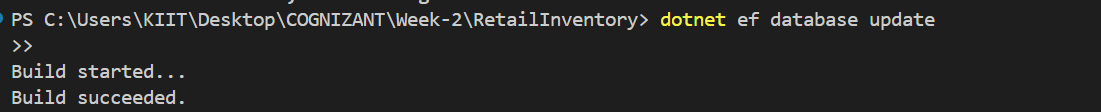
1. Install EF Core CLI (if not already):



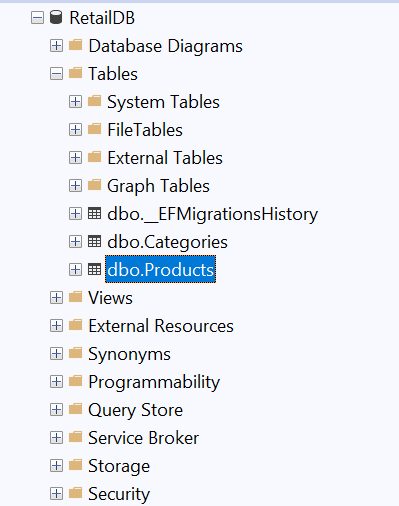
2. Create Initial Migration:



3. Apply Migration to Create Database:



4. Verify in SQL Server:



Lab 4: Inserting Initial Data into the Database

1. Insert Data in Program.cs:

using System;

using System.Threading.Tasks;

using RetailInventory.Models;

class Program

{

static async Task Main(string[] args)

{

using var context = new AppDbContext();

// Insert data

var electronics = new Category { Name = "Electronics" };

var groceries = new Category { Name = "Groceries" };

await context.Categories.AddRangeAsync(electronics, groceries);

var product1 = new Product { Name = "Laptop", Price = 75000, Category = electronics };

var product2 = new Product { Name = "Rice Bag", Price = 1200, Category = groceries };

await context.Products.AddRangeAsync(product1, product2);

await context.SaveChangesAsync();

Console.WriteLine("Data inserted successfully.\n");

// Fetch and display data

await DataFetcher.DisplayAllProductsAsync(context);

Console.WriteLine();

await DataFetcher.DisplayProductByIdAsync(context, 1);

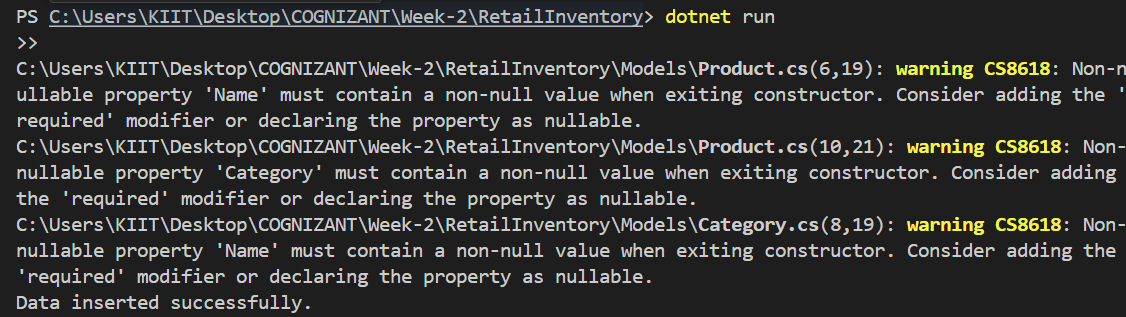
Console.WriteLine();

await DataFetcher.DisplayExpensiveProductAsync(context);

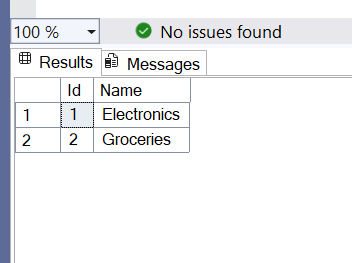
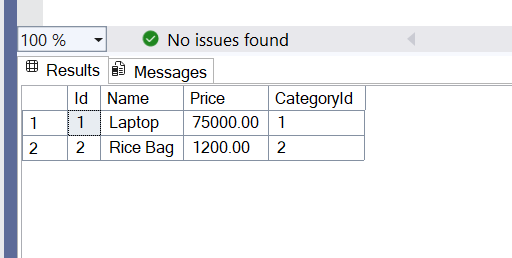
}

}

2. Run the App:



3. Verify in SQL Server:



Lab 5: Retrieving Data from the Database

1. Retrieve All Products:

using System;

using System.Threading.Tasks;

using System.Linq;

using RetailInventory.Models;

using Microsoft.EntityFrameworkCore;

public static class DataFetcher

{

public static async Task DisplayAllProductsAsync(AppDbContext context)

{

var products = await context.Products.Include(p => p.Category).ToListAsync();

foreach (var p in products)

{

Console.WriteLine($"{p.Name} - ₹{p.Price} - Category: {p.Category?.Name}");

}

}

public static async Task DisplayProductByIdAsync(AppDbContext context, int id)

{

var product = await context.Products.FindAsync(id);

if (product != null)

{

Console.WriteLine($"Found: {product.Name} - ₹{product.Price}");

}

else

{

Console.WriteLine("Product not found.");

}

}

public static async Task DisplayExpensiveProductAsync(AppDbContext context)

{

var expensive = await context.Products

.Include(p => p.Category)

.FirstOrDefaultAsync(p => p.Price > 50000);

if (expensive != null)

{

Console.WriteLine($"Expensive: {expensive.Name} - ₹{expensive.Price} - Category: {expensive.Category?.Name}");

}

else

{

Console.WriteLine("No expensive product found.");

}

}

}

2. Find by ID:

public static async Task DisplayProductByIdAsync(AppDbContext context, int id)

{

var product = await context.Products.FindAsync(id);

if (product != null)

{

Console.WriteLine($"Found: {product.Name} - ₹{product.Price}");

}

else

{

Console.WriteLine("Product not found.");

}

}

3. FirstOrDefault with Condition:

public static async Task DisplayExpensiveProductAsync(AppDbContext context)

{

var expensive = await context.Products

.Include(p => p.Category)

.FirstOrDefaultAsync(p => p.Price > 50000);

if (expensive != null)

{

Console.WriteLine($"Expensive: {expensive.Name} - ₹{expensive.Price} - Category: {expensive.Category?.Name}");

}

else

{

Console.WriteLine("No expensive product found.");

}

}

