WEEK – 3

EF Core 8.0 Guided Hands-On Exercises

# Lab 1: Understanding ORM with a Retail Inventory System

**Scenario:**

You’re building an inventory management system for a retail store. The store wants to track products, categories, and stock levels in a SQL Server database.

**Objective:**

Understand what ORM is and how EF Core helps bridge the gap between C# objects and relational tables.

**Steps:**

1. **What is ORM?**
   * **Explain how ORM maps C# classes to database tables.**

ORM stands for Object-Relational Mapping. It's a technique that allows you to interact with a database using C# objects instead of writing raw SQL queries.

➤ How it works:

* Classes ⇄ Tables
* Properties ⇄ Columns
* Objects ⇄ Rows
  + **Benefits: Productivity, maintainability, and abstraction from SQL. Productivity**: Less boilerplate code; more focus on business logic. **Maintainability**: Centralized models make changes easier.

**Abstraction**: You interact with objects, not raw SQL or connection logic.

1. **EF Core vs EF Framework:**
   * EF Core is cross-platform, lightweight, and supports modern features like LINQ, async queries, and compiled queries.
   * EF Framework (EF6) is Windows-only and more mature but less flexible.

|  |  |  |
| --- | --- | --- |
| **Feature** | **EF Core** | **Entity Framework (EF6)** |
| Platform | Cross-platform (.NET Core) | Windows-only (.NET Framework) |
| Lightweight | Yes | No |
| Performance | Improved with compiled queries | Slower in comparison |
| LINQ and Async | Full support | Limited async |
| JSON Column Mapping | Supported (EF Core 8.0+) | Not supported |
| Flexibility | Modular, extensible | Monolithic |

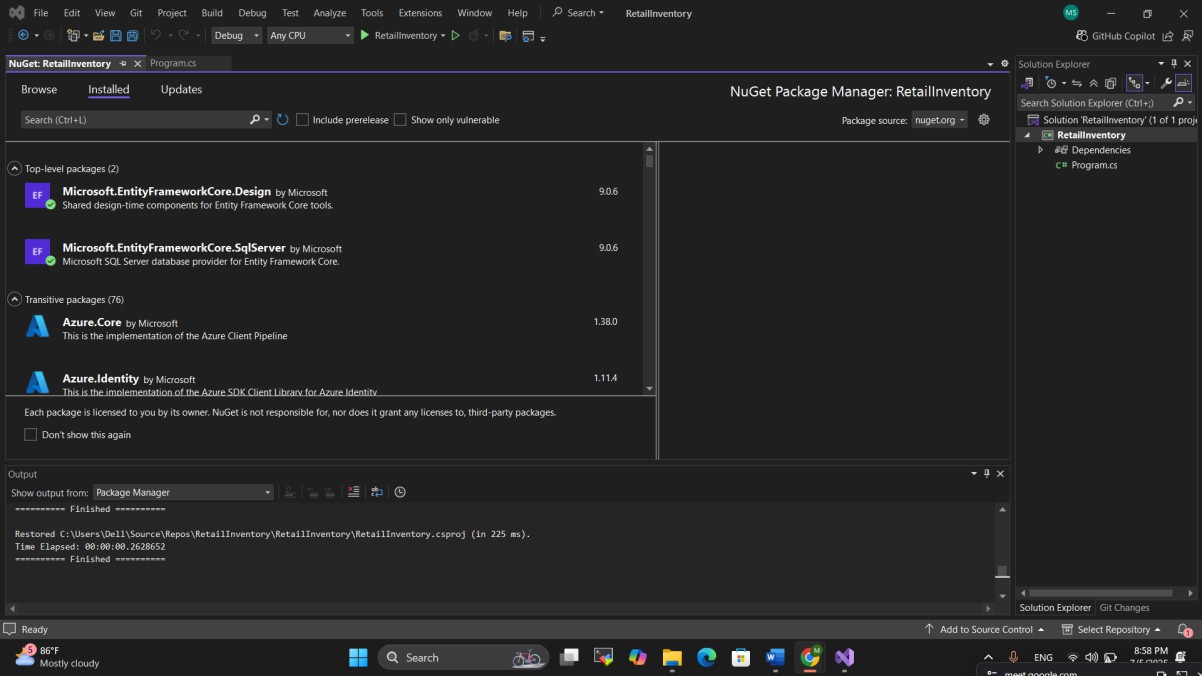
1. **EF Core 8.0 Features:**
   * JSON column mapping.
   * Store nested JSON data inside a column and query it natively.
   * Improved performance with compiled models.
   * Interceptors and better bulk operations.
   * Better support for updating large amounts of data efficiently.
   * Hook into database commands for logging, auditing, or modifying behavior.
2. **Create a .NET Console App:**

dotnet new console -n RetailInventory cd RetailInventory

1. **Install EF Core Packages:**

dotnet add package Microsoft.EntityFrameworkCore.SqlServer dotnet add package Microsoft.EntityFrameworkCore.Design

**OUTPUT :**



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

**Scenario:**

The retail store wants to store product and category data in SQL Server.

**Objective:**

Configure DbContext and connect to SQL Server.

**Steps:**

1. **Create Models:**

public class Category {

public int Id { get; set; } public string Name { get; set; } public List Products { get; set; }

}

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; }

}

1. **Create AppDbContext:**

public class AppDbContext : DbContext { public DbSet Products { get; set; } public DbSet Categories { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuild er)

{

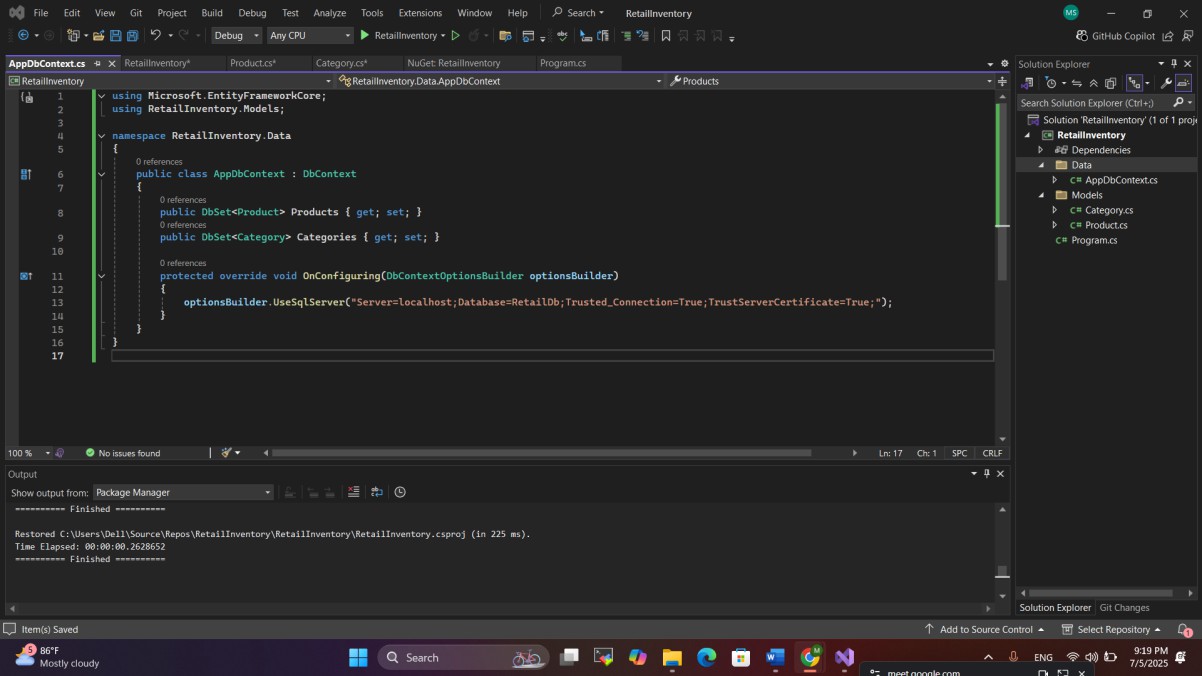
optionsBuilder.UseSqlServer("Your\_Connection\_String\_Here");

}

}

1. **Add Connection String in appsettings.json (optional for ASP.NET Core).**

**OUTPUT :**

****

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

**Scenario:**

The retail store's database needs to be created based on the models you've defined. You’ll use EF Core CLI to generate and apply migrations.

**Objective:**

Learn how to use EF Core CLI to manage database schema changes.

**Steps:**

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

dotnet tool install --global dotnet-ef

1. **Create Initial Migration:**

dotnet ef migrations add InitialCreate This generates a Migrations folder with code that represents the schema.

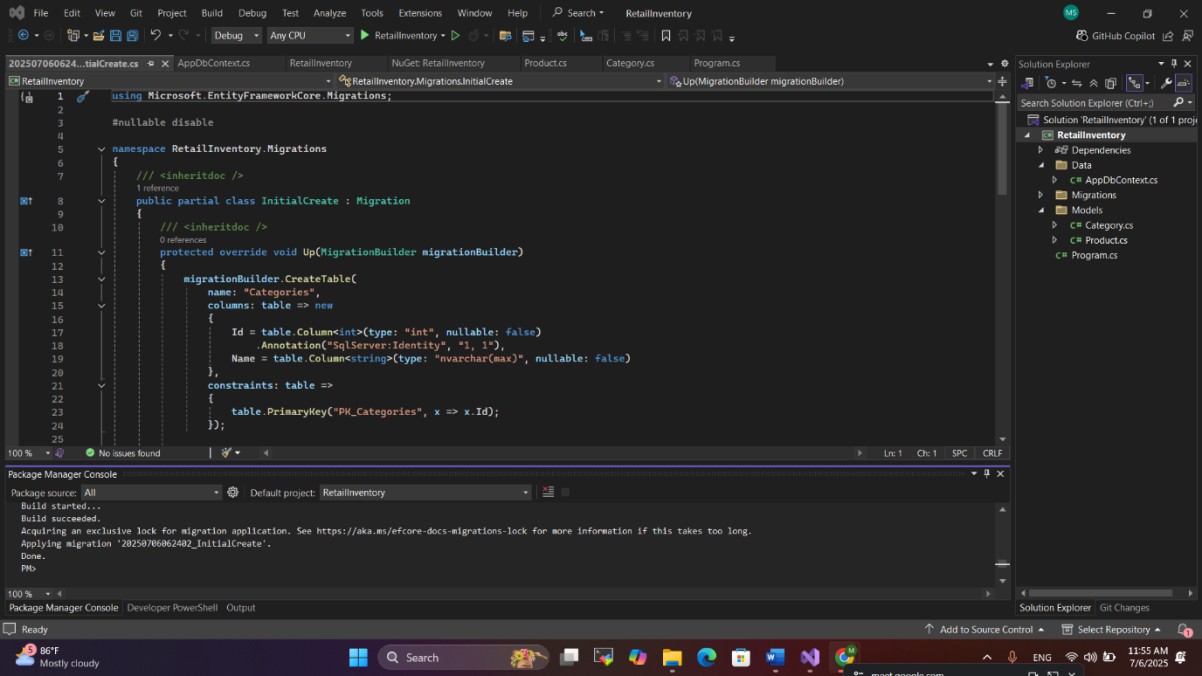
1. **Apply Migration to Create Database:**

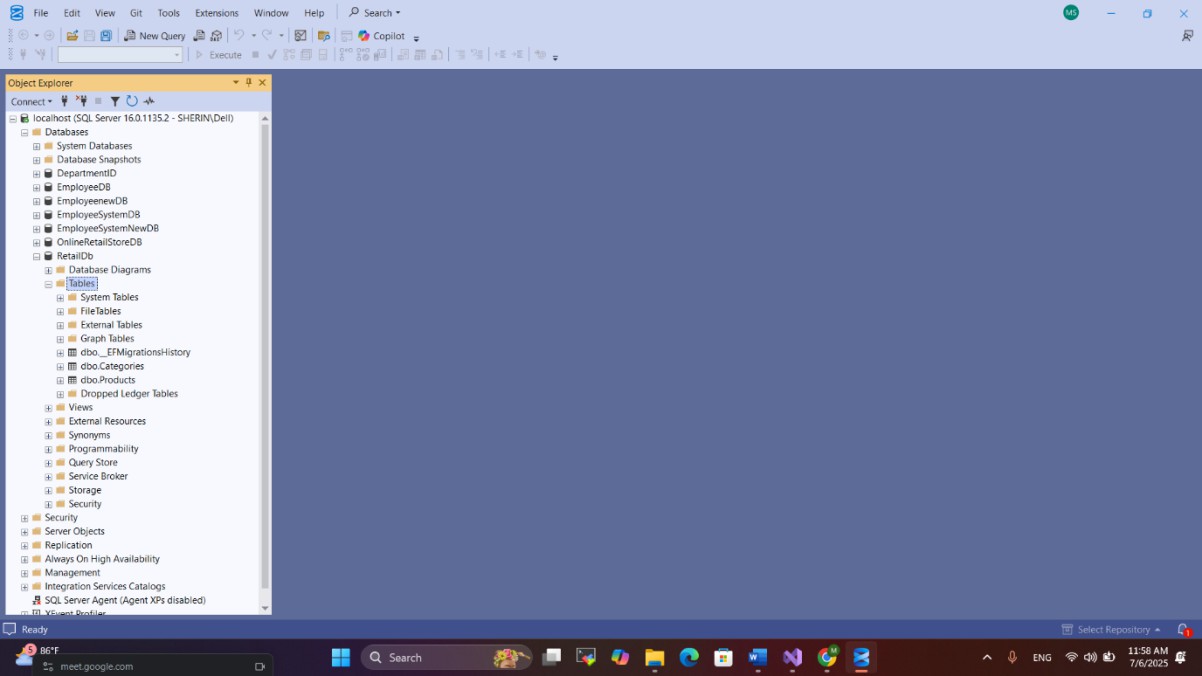
dotnet ef database update

1. **Verify in SQL Server:**

Open SQL Server Management Studio (SSMS) or Azure Data Studio and confirm that tables Products and Categories are created.

**OUTPUT :**

****



# Lab 4: Inserting Initial Data into the Database

**Scenario:**

The store manager wants to add initial product categories and products to the system.

**Objective:**

Use EF Core to insert records using AddAsync and SaveChangesAsync.

**Steps:**

1. **Insert Data in Program.cs:**

using var context = new AppDbContext();

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 = electro nics };

var product2 = new Product { Name = "Rice Bag", Price = 1200, Category = groceri es

};

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

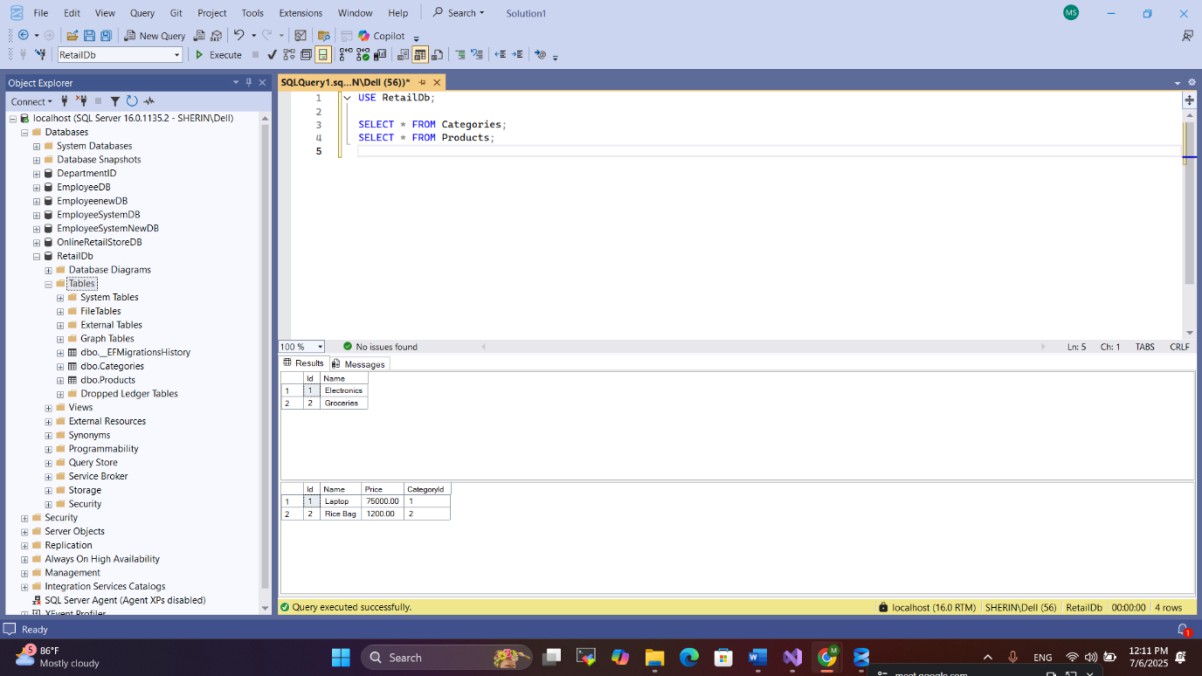
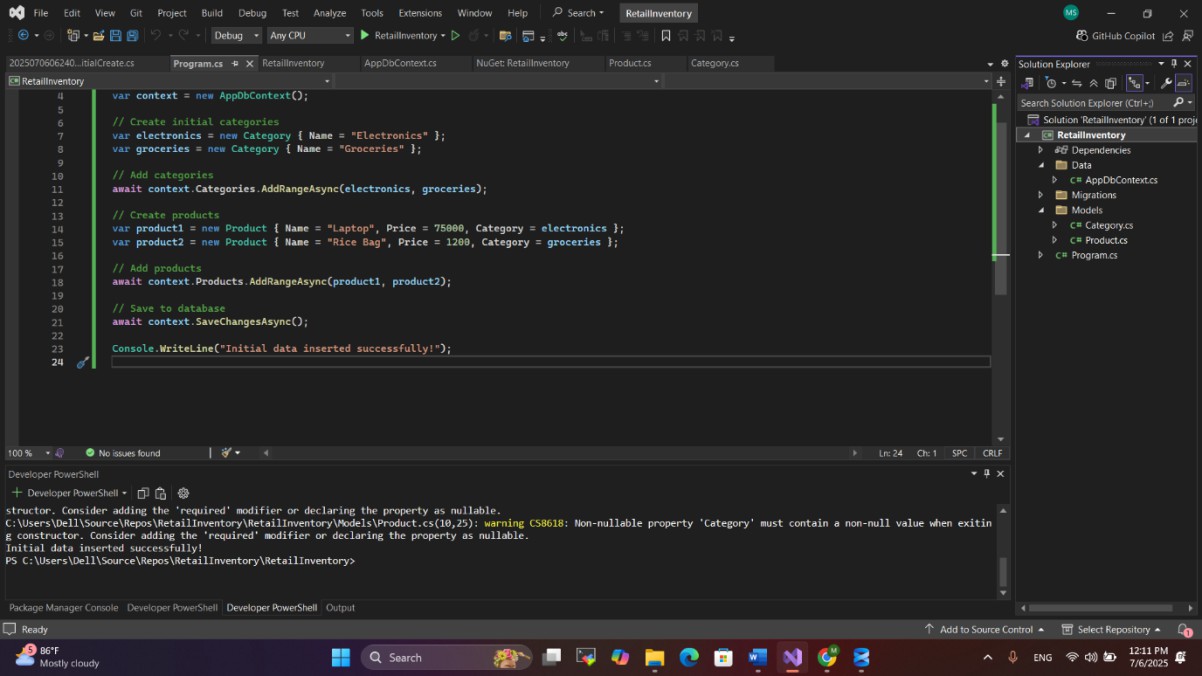
1. **Run the App:**

dotnet run

1. **Verify in SQL Server:**

Check that the data is inserted correctly.

**OUTPUT :**

****

# Lab 5: Retrieving Data from the Database

**Scenario:**

The store wants to display product details on the dashboard.

**Objective:**

Use Find, FirstOrDefault, and ToListAsync to retrieve data.

**Steps:**

1. **Retrieve All Products:**

var products = await context.Products.ToListAsync(); foreach (var p in products)

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

1. **Find by ID:**

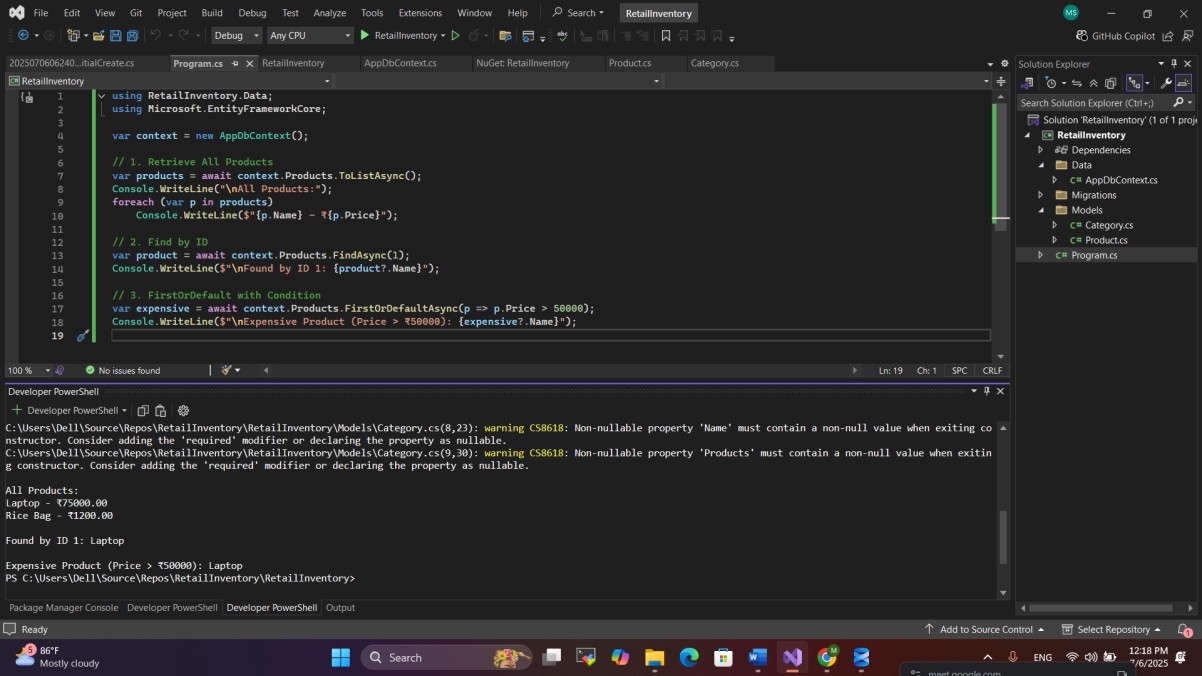
var product = await context.Products.FindAsync(1); Console.WriteLine($"Found: {product?.Name}");

1. **FirstOrDefault with Condition:**

var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 5000 0);

Console.WriteLine($"Expensive: {expensive?.Name}");

**OUTPUT :**

****

# Lab 6: Updating and Deleting Records

**Scenario:**

The store updates product prices and removes discontinued items.

**Objective:**

Update and delete records using EF Core.

**Steps:**

1. **Update a Product:**

var product = await context.Products.FirstOrDefaultAsync(p => p.Name == "Lapt op"); if (product != null) {

product.Price = 70000;

await context.SaveChangesAsync();

}

1. **Delete a Product:**

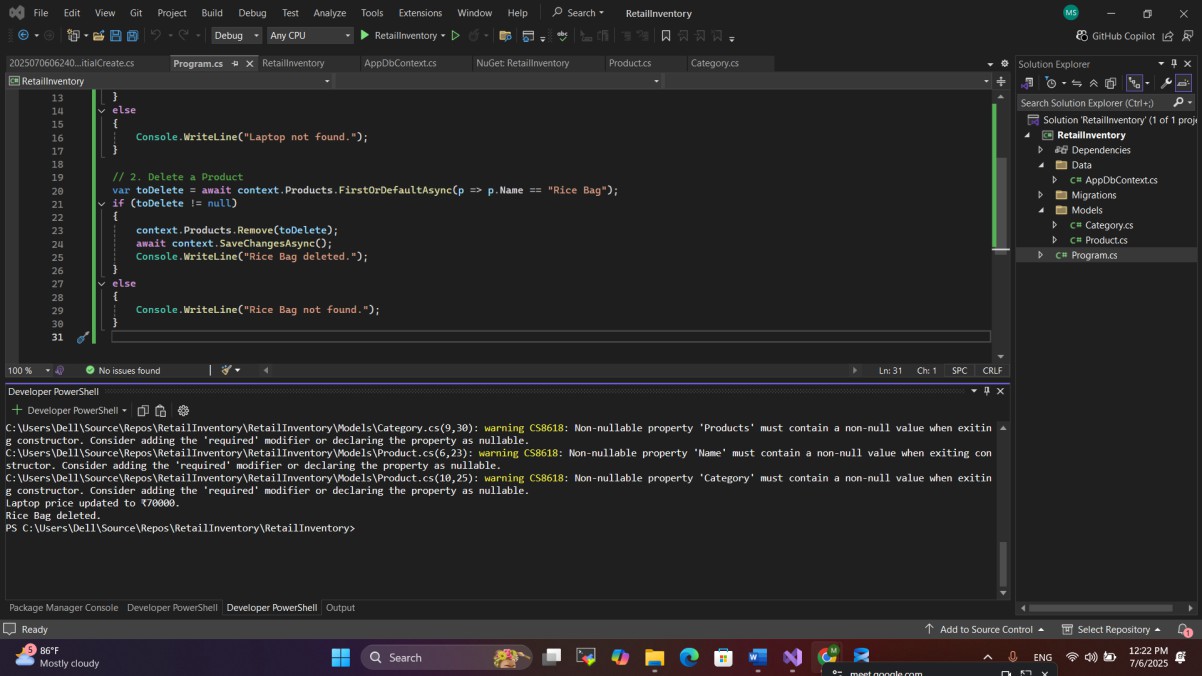
var toDelete = await context.Products.FirstOrDefaultAsync(p => p.Name == "Rice Bag");

if (toDelete != null) {

context.Products.Remove(toDelete); await context.SaveChangesAsync();

}

# OUTPUT :



**Lab 7: Writing Queries with LINQ**

**Scenario:**

The store wants to filter and sort products for reporting.

**Objective:**

Use Where, Select, OrderBy, and project into DTOs.

**Steps:**

1. **Filter and Sort:**

var filtered = await context.Products .Where(p => p.Price > 1000)

.OrderByDescending(p => p.Price)

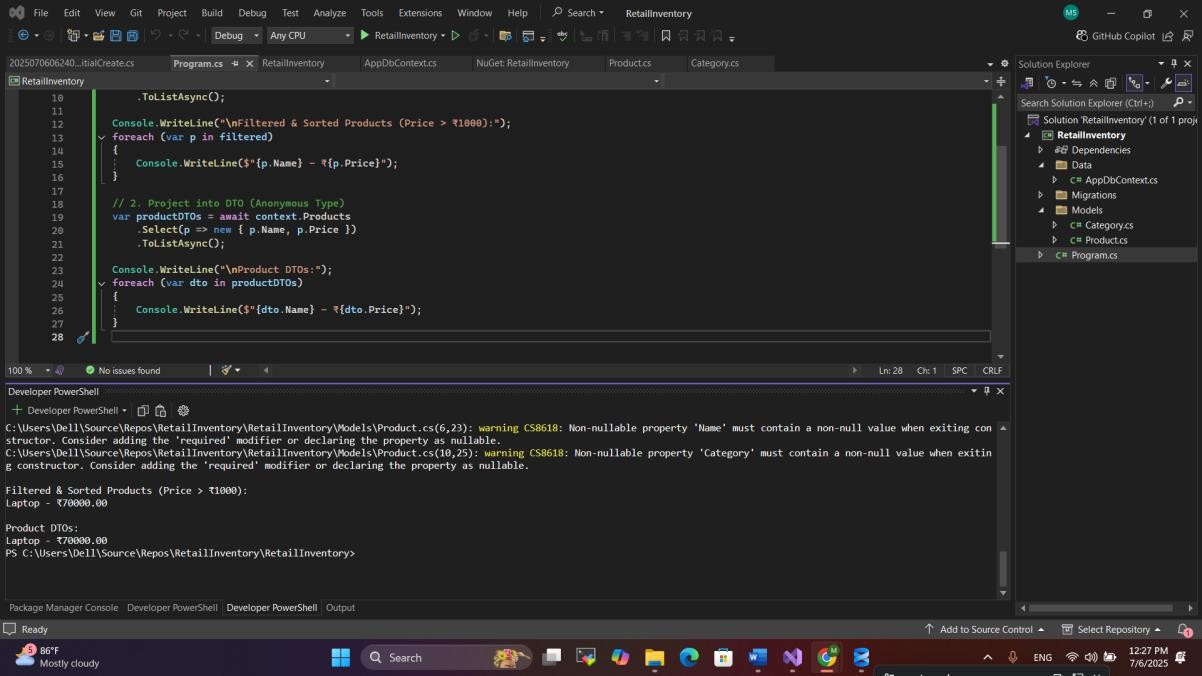
.ToListAsync();

1. **Project into DTO:**

var productDTOs = await context.Products

.Select(p => new { p.Name, p.Price })

.ToListAsync();

******OUTPUT:**