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.

2. **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

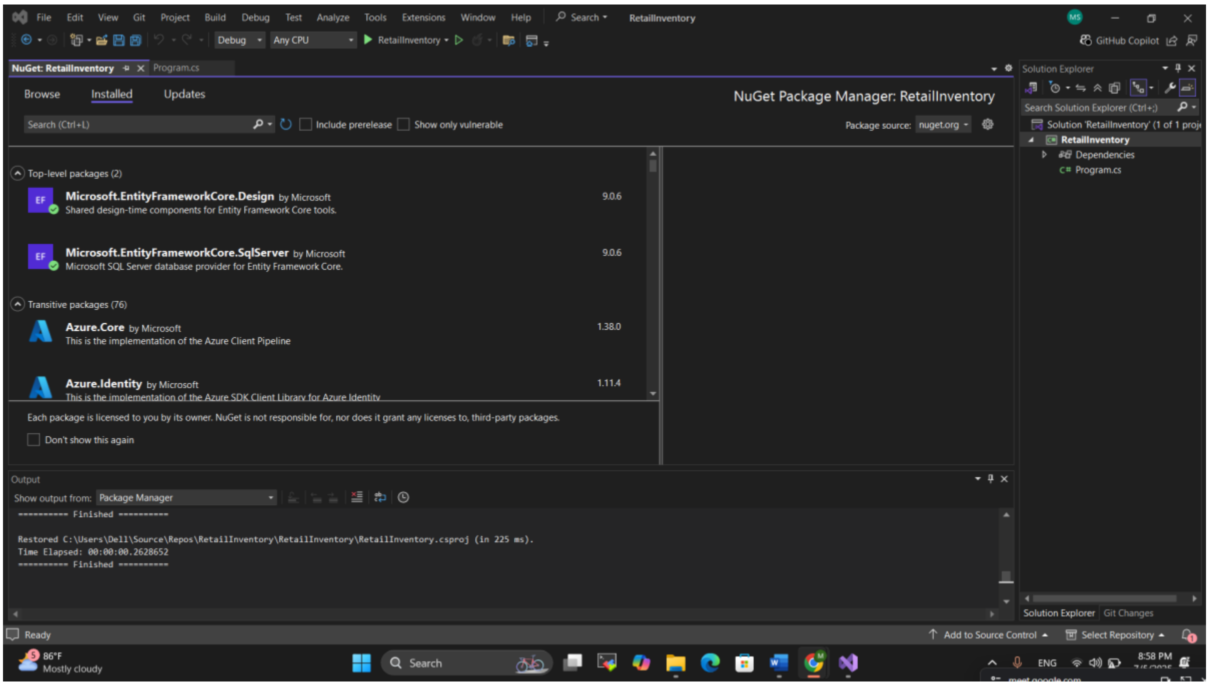
3. **EF Core 8.0 Features:**
    - JSON column mapping.
    - Store nested JSON data inside a column and query it natively.
    - Improved performance with compiled models.
    - Interceptors and better bulk operations.
    - Better support for updating large amounts of data efficiently.
    - Hook into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

```
dotnet new console -n RetailInventory cd
RetailInventory
```

5. **Install EF Core Packages:**

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

OUTPUT :



NuGet Package Manager: RetailInventory

□ Include prerelease □ Show only vulnerable

Package source: nuget.org

Top-level packages (2)

Microsoft.EntityFrameworkCore.Design by Microsoft Shared design-time components for Entity Framework Core tools.	9.0.6
Microsoft.EntityFrameworkCore.SqlServer by Microsoft Microsoft SQL Server database provider for Entity Framework Core.	9.0.6

Transitive packages (76)

Azure.Core by Microsoft This is the implementation of the Azure Client Pipeline	1.38.0
Azure.Identity by Microsoft This is the implementation of the Azure SDK Client Library for Azure Identity	1.11.4

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

□ Don't show this again

Output

Show output from: Package Manager

```
========== Finished ==========

Restored C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\RetailInventory.csproj (in 225 ms).
Time Elapsed: 00:00:00.2628652
========== Finished ==========
```

Ready

86°F
Mostly cloudy

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:**

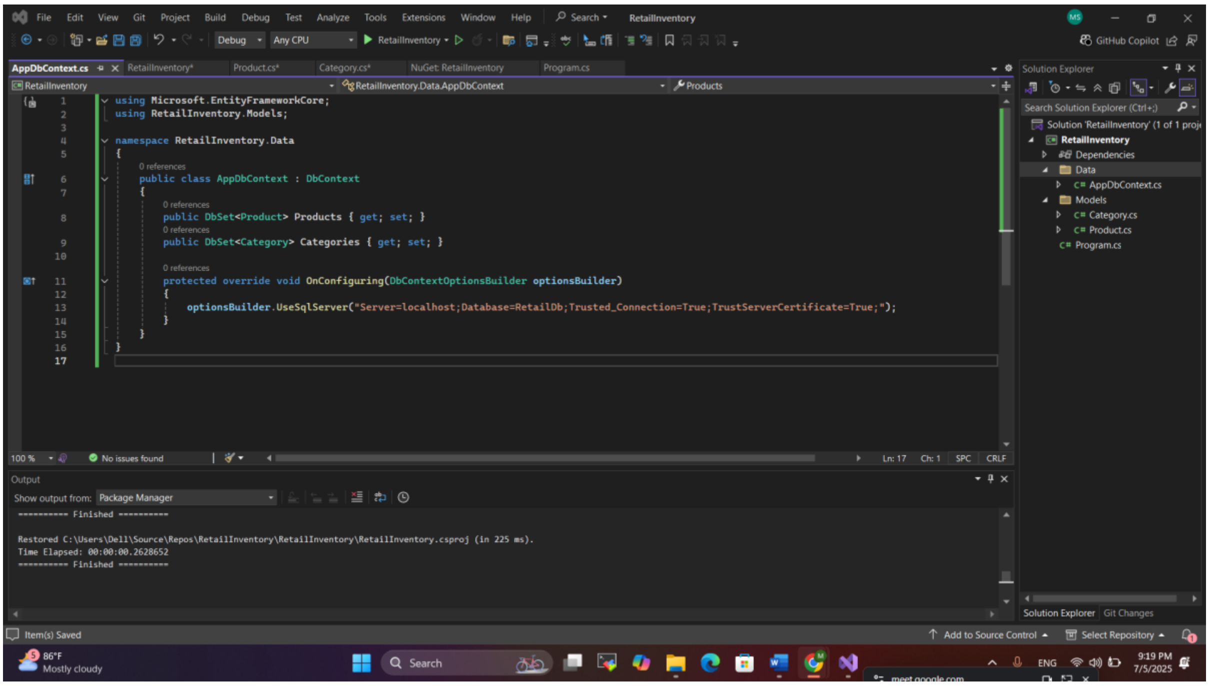
```
public class Category {
   public int Id { get; set; }     public
string Name { get; set; }     public
List Products { get; set; }
}
public class Product {
   public int Id { get; set; }     public
string Name { get; set; }     public
decimal Price { get; set; }     public int
CategoryId { get; set; }     public
Category Category { get; set; }
}
```

2. **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");
   }
}
```

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

OUTPUT :



```
using Microsoft.EntityFrameworkCore;
using RetailInventory.Models;

namespace RetailInventory.Data
{
    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=localhost;Database=RetailDb;Trusted_Connection=True;TrustServerCertificate=True;");
        }
    }
}
```

Show output from: Package Manager

```
========== Finished ==========

Restored C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\RetailInventory.csproj (in 225 ms).
Time Elapsed: 00:00:00.2628652
========== Finished ==========
```

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
```

2. **Create Initial Migration:**

```
dotnet ef migrations add InitialCreate
```

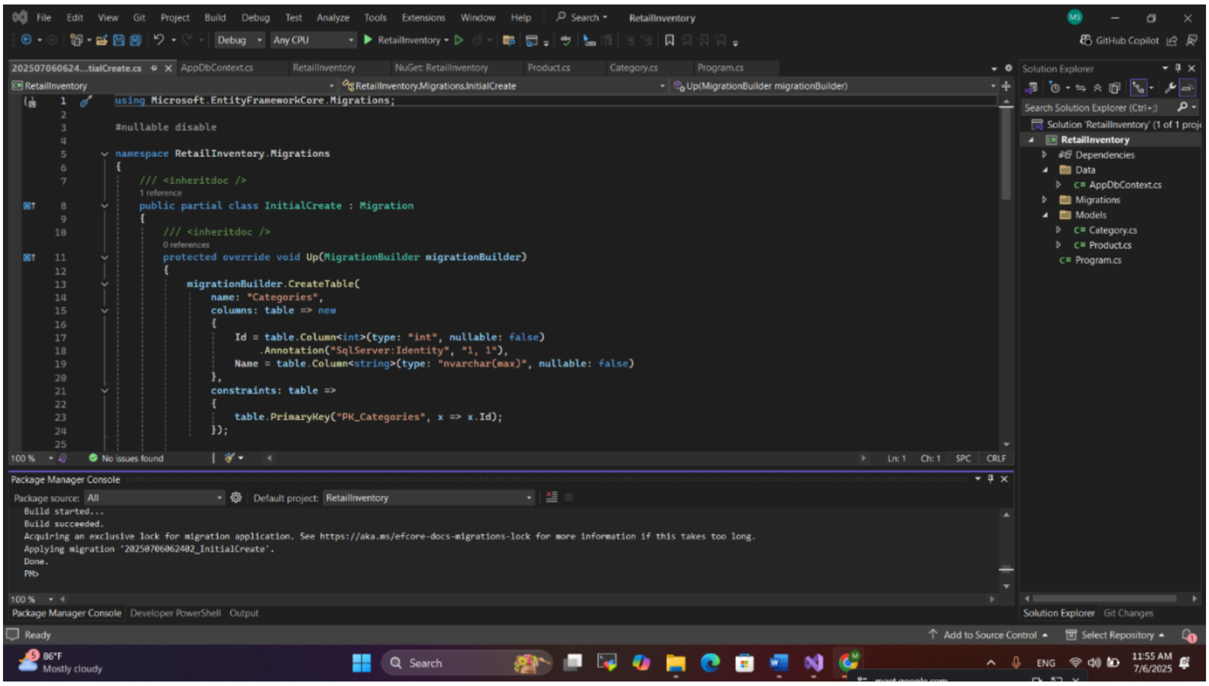
This generates a Migrations folder with code that represents the schema.

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

```
dotnet ef database update
```

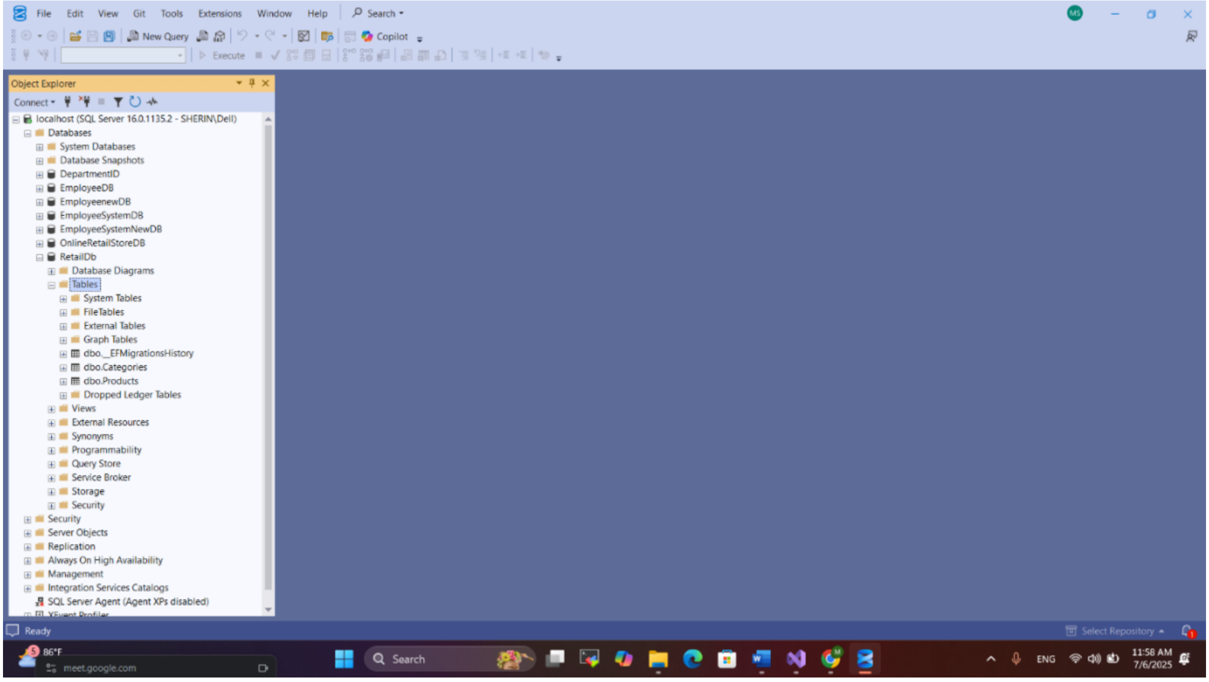
4. **Verify in SQL Server:**
   Open SQL Server Management Studio (SSMS) or Azure Data Studio and confirm that tables Products and Categories are created.

OUTPUT :



```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace RetailInventory.Migrations
{
    /// <inheritdoc />
    public partial class InitialCreate : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Categories",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Name = table.Column<string>(type: "nvarchar(max)", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Categories", x => x.Id);
                });
```

```
Build started...
Build succeeded.
Acquiring an exclusive lock for migration application. See https://aka.ms/efcore-docs-migrations-lock for more information if this takes too long.
Applying migration '20250706062402_InitialCreate'.
Done.
PM>
```



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:**

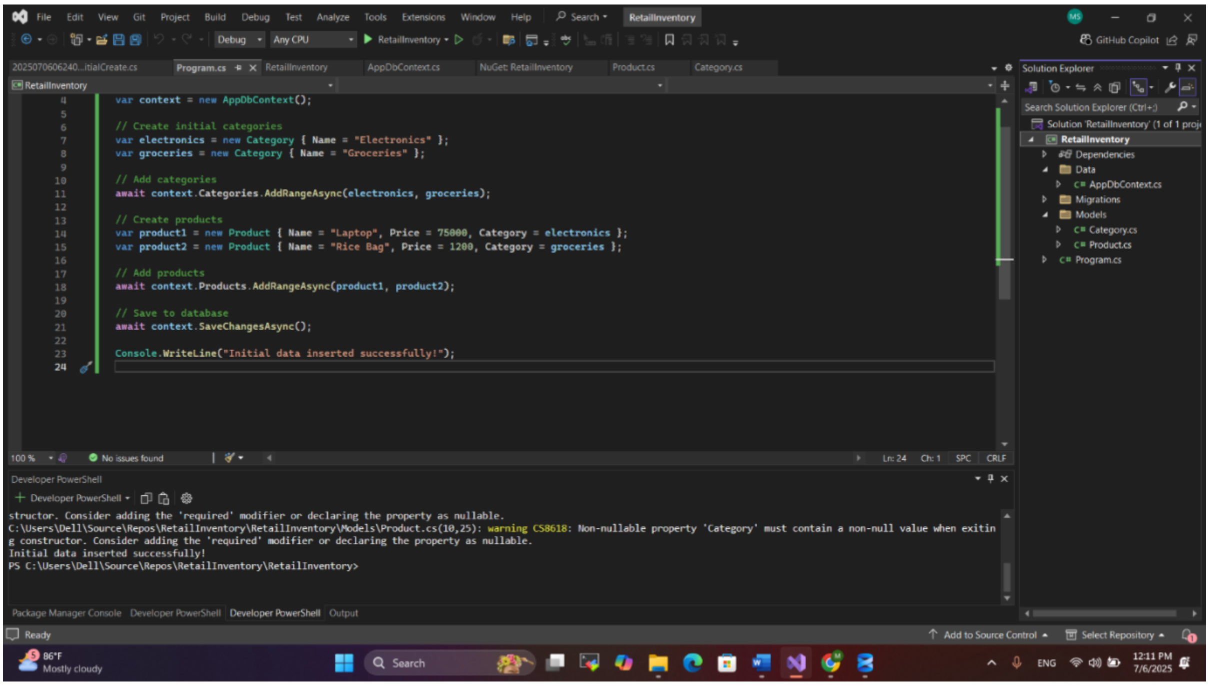
```
using var context = new AppDbContext();

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();
```

2. **Run the App:**

```
dotnet run
```

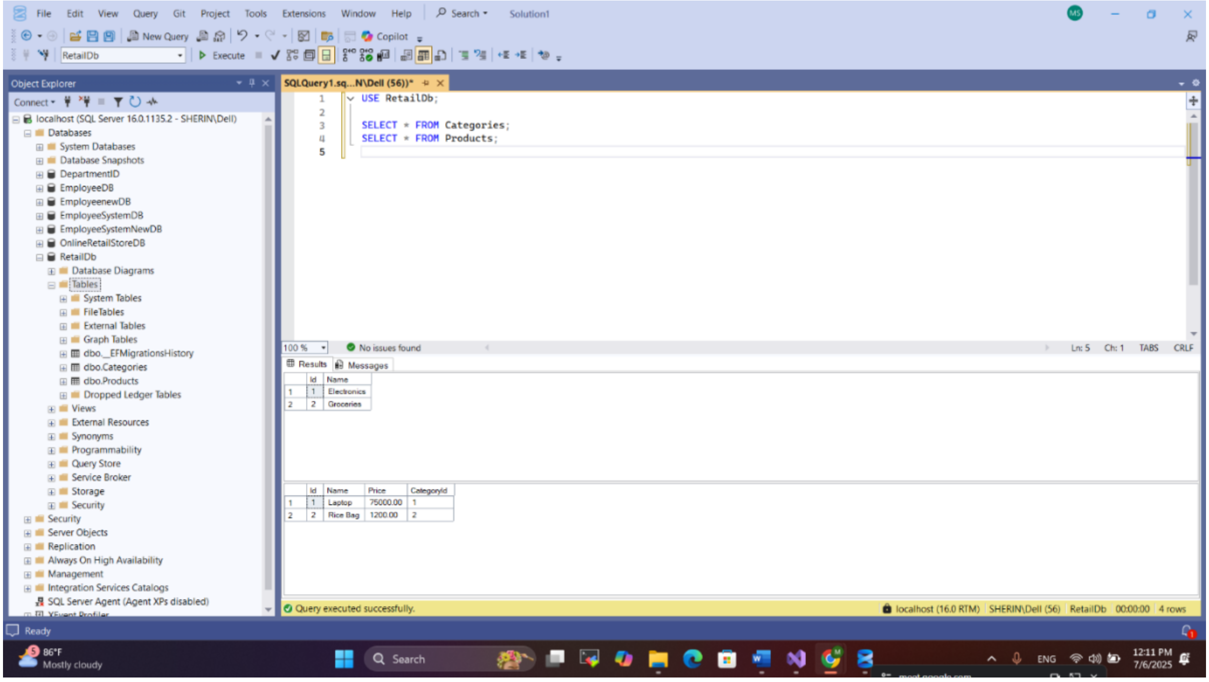
3. **Verify in SQL Server:**
   Check that the data is inserted correctly.

OUTPUT :



```
var context = new AppDbContext();

// Create initial categories
var electronics = new Category { Name = "Electronics" };
var groceries = new Category { Name = "Groceries" };

// Add categories
await context.Categories.AddRangeAsync(electronics, groceries);

// Create products
var product1 = new Product { Name = "Laptop", Price = 75000, Category = electronics };
var product2 = new Product { Name = "Rice Bag", Price = 1200, Category = groceries };

// Add products
await context.Products.AddRangeAsync(product1, product2);

// Save to database
await context.SaveChangesAsync();

Console.WriteLine("Initial data inserted successfully!");
```

```
structor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Product.cs(10,25): warning CS8618: Non-nullable property 'Category' must contain a non-null value when exitin
g constructor. Consider adding the 'required' modifier or declaring the property as nullable.
Initial data inserted successfully!
PS C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory>
```



```
USE RetailDb;

SELECT * FROM Categories;
SELECT * FROM Products;
```

	Id	Name
1	1	Electronics
2	2	Groceries

	Id	Name	Price	CategoryId
1	1	Laptop	75000.00	1
2	2	Rice Bag	1200.00	2

Query executed successfully.

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}");
```

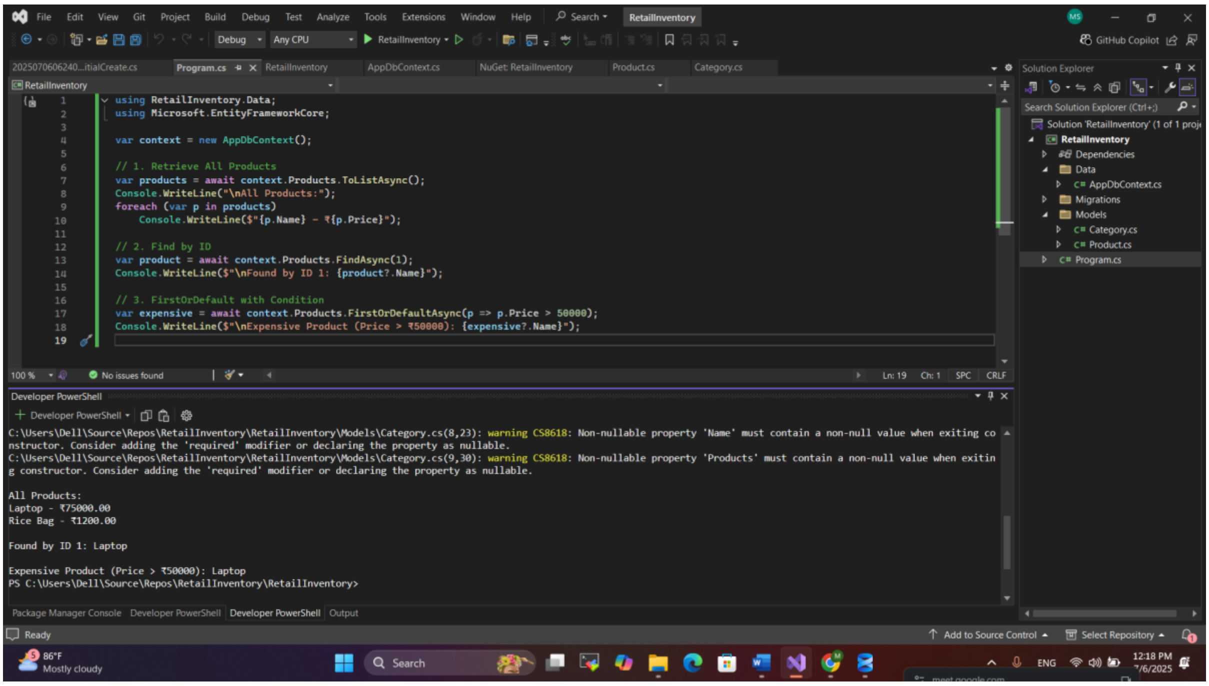
2. **Find by ID:**

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

3. **FirstOrDefault with Condition:**

```
var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 5000
0);
Console.WriteLine($"Expensive: {expensive?.Name}");
```

OUTPUT :



```
using RetailInventory.Data;
using Microsoft.EntityFrameworkCore;

var context = new AppDbContext();

// 1. Retrieve All Products
var products = await context.Products.ToListAsync();
Console.WriteLine("\nAll Products:");
foreach (var p in products)
    Console.WriteLine($"{p.Name} - ₹{p.Price}");

// 2. Find by ID
var product = await context.Products.FindAsync(1);
Console.WriteLine($"\nFound by ID 1: {product?.Name}");

// 3. FirstOrDefault with Condition
var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 50000);
Console.WriteLine($"\nExpensive Product (Price > ₹50000): {expensive?.Name}");
```

```
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Category.cs(8,23): warning CS8618: Non-nullable property 'Name' must contain a non-null value when exiting co
nstructor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Category.cs(9,30): warning CS8618: Non-nullable property 'Products' must contain a non-null value when exitin
g constructor. Consider adding the 'required' modifier or declaring the property as nullable.

All Products:
Laptop - ₹75000.00
Rice Bag - ₹1200.00

Found by ID 1: Laptop

Expensive Product (Price > ₹50000): Laptop
PS C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory>
```

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:

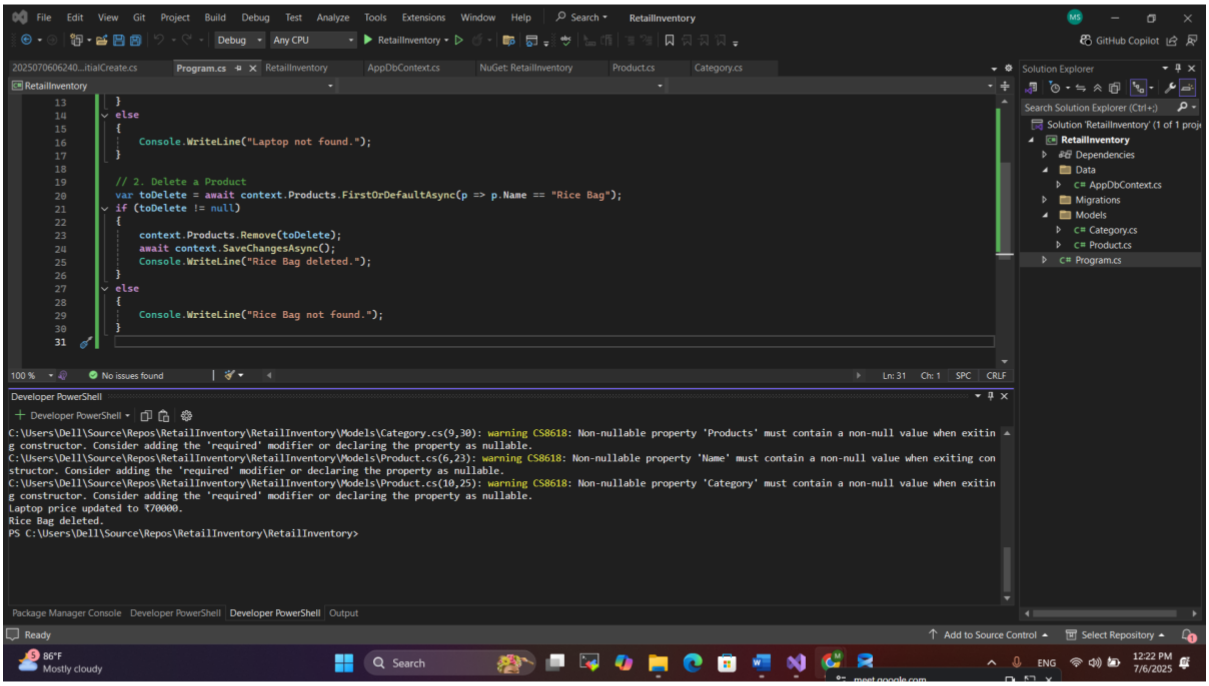
1. **Update a Product:**

```
var product = await context.Products.FirstOrDefaultAsync(p => p.Name == "Lapt op");
if (product != null) {
product.Price = 70000;
   await context.SaveChangesAsync();
}
```

2. **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 :



```
    }
    else
    {
        Console.WriteLine("Laptop not found.");
    }

    // 2. Delete a Product
    var toDelete = await context.Products.FirstOrDefaultAsync(p => p.Name == "Rice Bag");
    if (toDelete != null)
    {
        context.Products.Remove(toDelete);
        await context.SaveChangesAsync();
        Console.WriteLine("Rice Bag deleted.");
    }
    else
    {
        Console.WriteLine("Rice Bag not found.");
    }
```

```
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Category.cs(9,30): warning CS8618: Non-nullable property 'Products' must contain a non-null value when exitin
g constructor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Product.cs(6,23): warning CS8618: Non-nullable property 'Name' must contain a non-null value when exiting con
structor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Product.cs(10,25): warning CS8618: Non-nullable property 'Category' must contain a non-null value when exitin
g constructor. Consider adding the 'required' modifier or declaring the property as nullable.
Laptop price updated to ₹70000.
Rice Bag deleted.
PS C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory>
```

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:

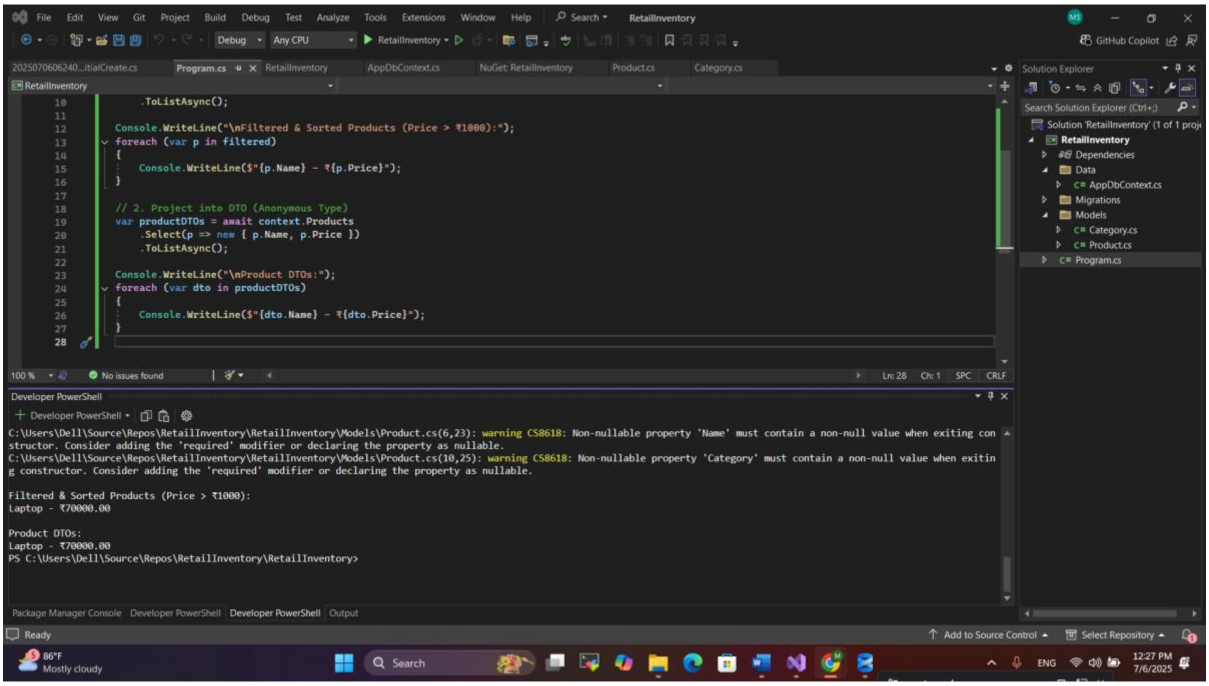
1. **Filter and Sort:**

```
var filtered = await context.Products     .Where(p => p.Price > 1000)
    .OrderByDescending(p => p.Price)
    .ToListAsync();
```

2. **Project into DTO:**

```
var productDTOs = await context.Products
    .Select(p => new { p.Name, p.Price })
    .ToListAsync();
```

OUTPUT:



```csharp
    .ToListAsync();

Console.WriteLine("\nFiltered & Sorted Products (Price > ₹1000):");
foreach (var p in filtered)
{
    Console.WriteLine($"{p.Name} - ₹{p.Price}");
}

// 2. Project into DTO (Anonymous Type)
var productDTOs = await context.Products
    .Select(p => new { p.Name, p.Price })
    .ToListAsync();

Console.WriteLine("\nProduct DTOs:");
foreach (var dto in productDTOs)
{
    Console.WriteLine($"{dto.Name} - ₹{dto.Price}");
}
```

```
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Product.cs(6,23): warning CS8618: Non-nullable property 'Name' must contain a non-null value when exiting con
structor. Consider adding the 'required' modifier or declaring the property as nullable.
C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory\Models\Product.cs(10,25): warning CS8618: Non-nullable property 'Category' must contain a non-null value when exitin
g constructor. Consider adding the 'required' modifier or declaring the property as nullable.

Filtered & Sorted Products (Price > ₹1000):
Laptop - ₹70000.00

Product DTOs:
Laptop - ₹70000.00
PS C:\Users\Dell\Source\Repos\RetailInventory\RetailInventory>
```