(WEEK - 3 HANDSON EXERCISE)

ADITYA RAJ

SUPERSET ID - 6363544

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.

Ans:

ORM maps object-oriented code (C# classes) to relational database structures (tables). This process is called object-relational mapping.

How Mapping Works in Entity Framework:

Class to Table:

Each C# class becomes a table in the database.

public class Student

{

public int StudentId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

}

Maps to this SQL Table:

CREATE TABLE Students (

StudentId INT PRIMARY KEY,

FirstName NVARCHAR(MAX),

LastName NVARCHAR(MAX)

);

Property maps to Column:

Each property of the class maps to a column in the table.

Primary Key Identification

EF automatically identifies a primary key if:

Property is named Id

Or: <ClassName>Id (e.g., StudentId in Student)

Relationships Mapping

EF uses navigation properties and conventions or attributes to define relationships.

One-to-Many Example:

public class Student

{

public int StudentId { get; set; }

public ICollection<Enrollment> Enrollments { get; set; }

}

public class Enrollment

{

public int EnrollmentId { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

}

This will create:

Students table

Enrollments table with StudentId as foreign key.

. DbContext as the Bridge

Your DbContext tells EF which classes to treat as entities (tables):

using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext

{

public DbSet<Student> Students { get; set; }

public DbSet<Enrollment> Enrollments { get; set; }

}

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

Ans:

1. Productivity

ORMs like Entity Framework help developers build data-driven applications faster by eliminating a lot of boilerplate code.

a ) Automatically generates SQL for common operations (INSERT, SELECT, UPDATE, DELETE).

b) Handles relationships and joins via navigation properties.

2. Maintainability

A ) EF keeps your data model in sync with your application logic, making code easier to understand and update.

B ) Strongly typed: C# compiler catches errors at compile time (e.g., mistyped column/property names).

C ) Centralized model definitions make it easier to track changes.

1. Easier to refactor your model classes (e.g., rename properties safely).
2. Manual SQL: Renaming a column = updating every raw SQL string.
3. : Rename a C# property, and EF migration will track it automatically.

3. Abstraction from SQL

Entity Framework lets you focus on business logic instead of query syntax.

A ) You write C# (LINQ) queries instead of SQL.

B ) You don't need deep SQL knowledge to work with databases.

C )EF handles connection pooling, transactions, and parameterization safely.

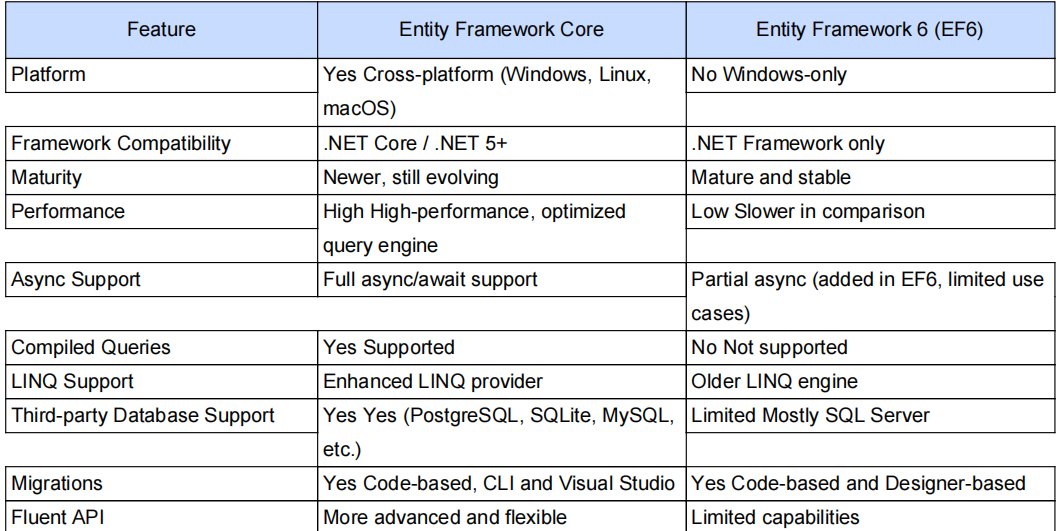
D )Reduces need for manual database access code (e.g., ADO.NET, SqlCommand).

3. 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.



4 ) EF Core 8.0 Features:

Ans :

JSON column mapping.

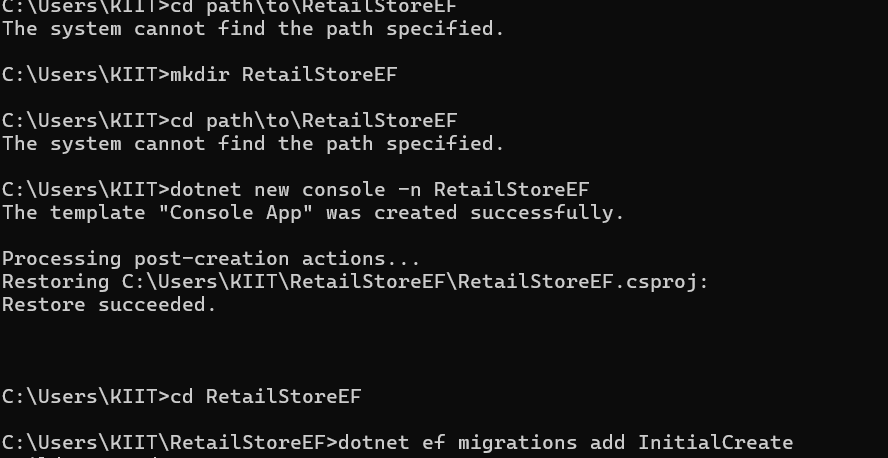
Improved performance with compiled models.

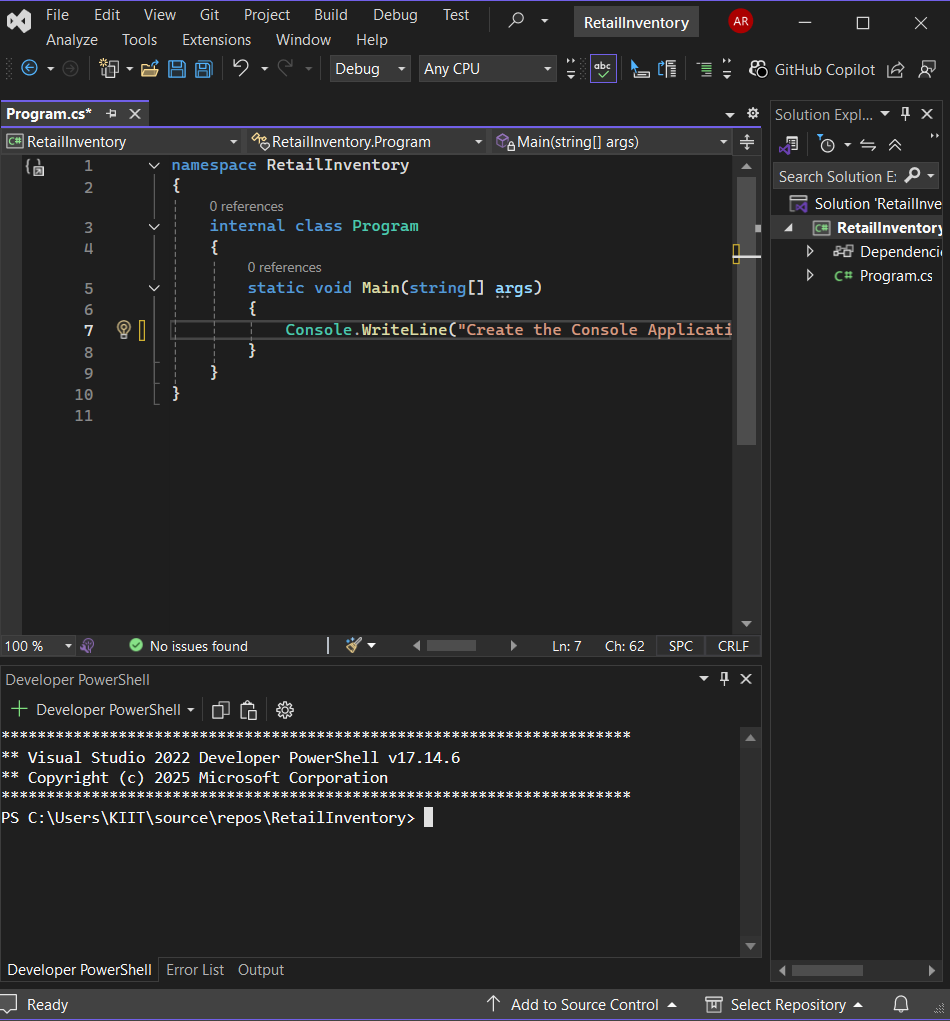
Interceptors and better bulk operations.

5. Create a .NET Console App:

dotnet new console -n RetailInventory

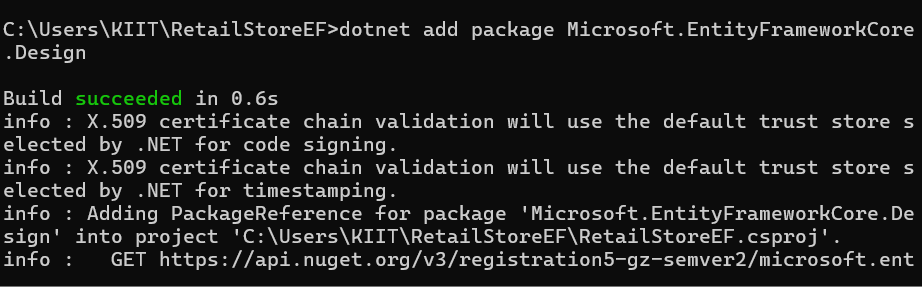
cd RetailInventory



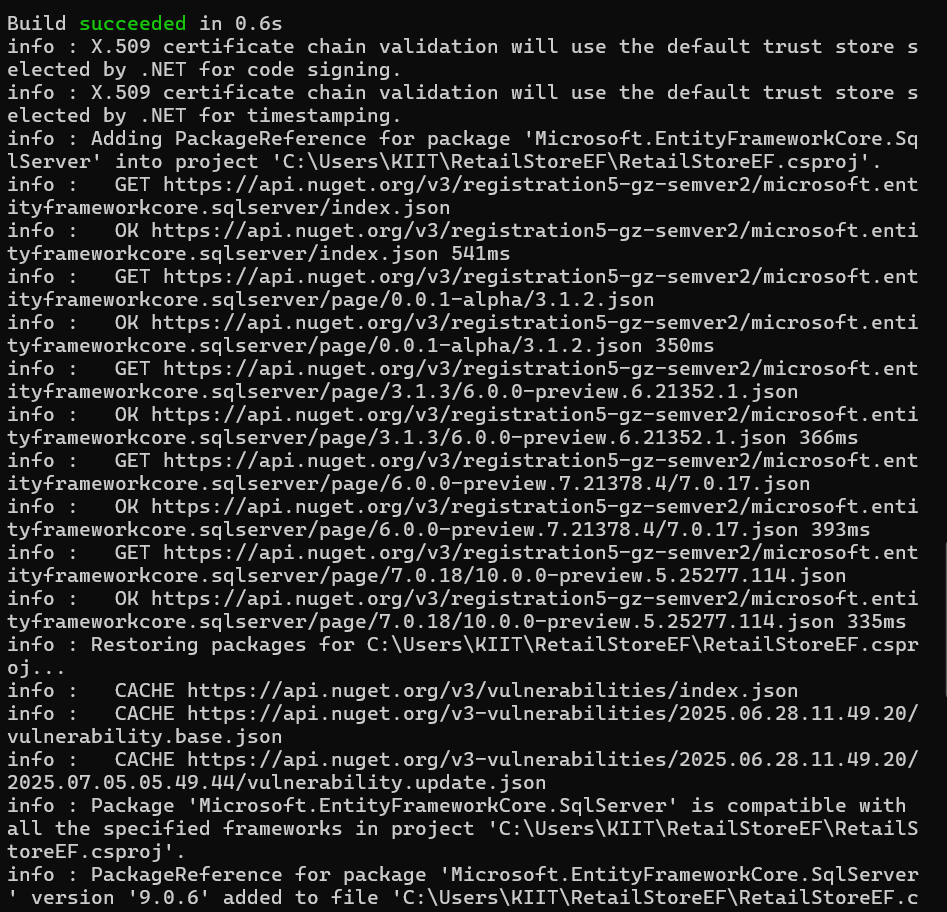


5. Install EF Core Packages:

dotnet add package Microsoft.EntityFrameworkCore.Design



dotnet add package Microsoft.EntityFrameworkCore.SqlServer



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; }

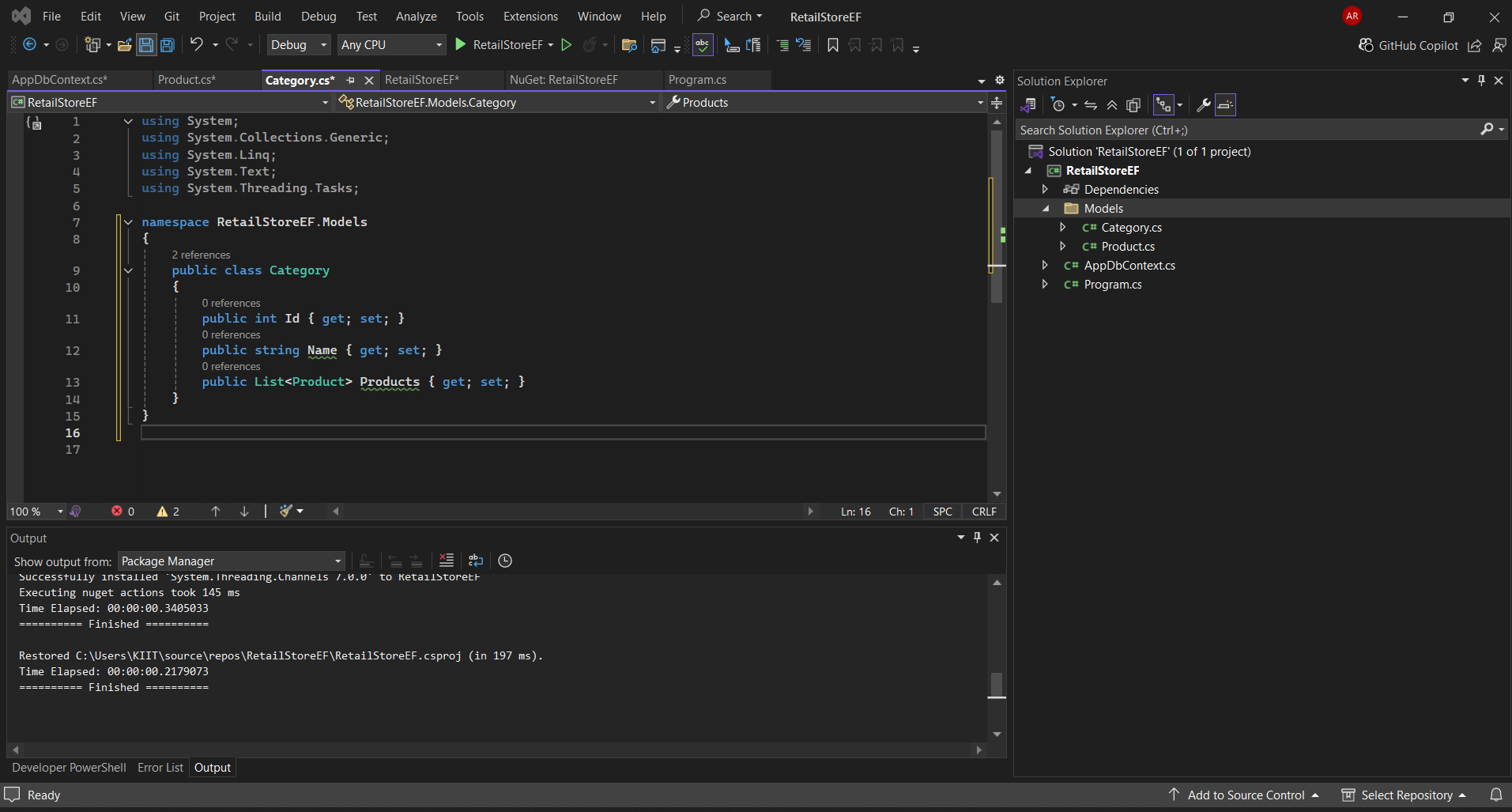
public string Name { get; set; }

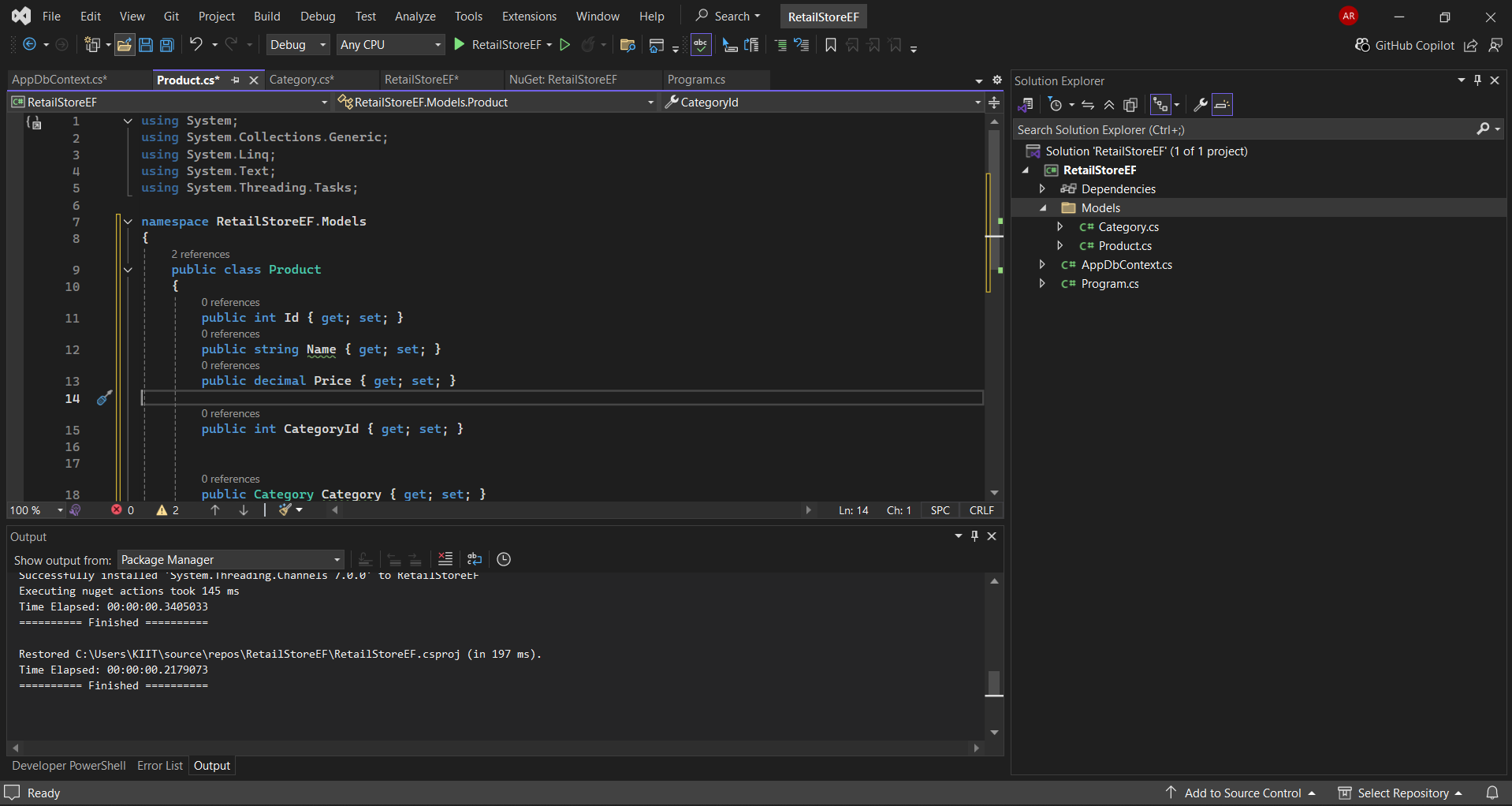
public decimal Price { get; set; }

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; }

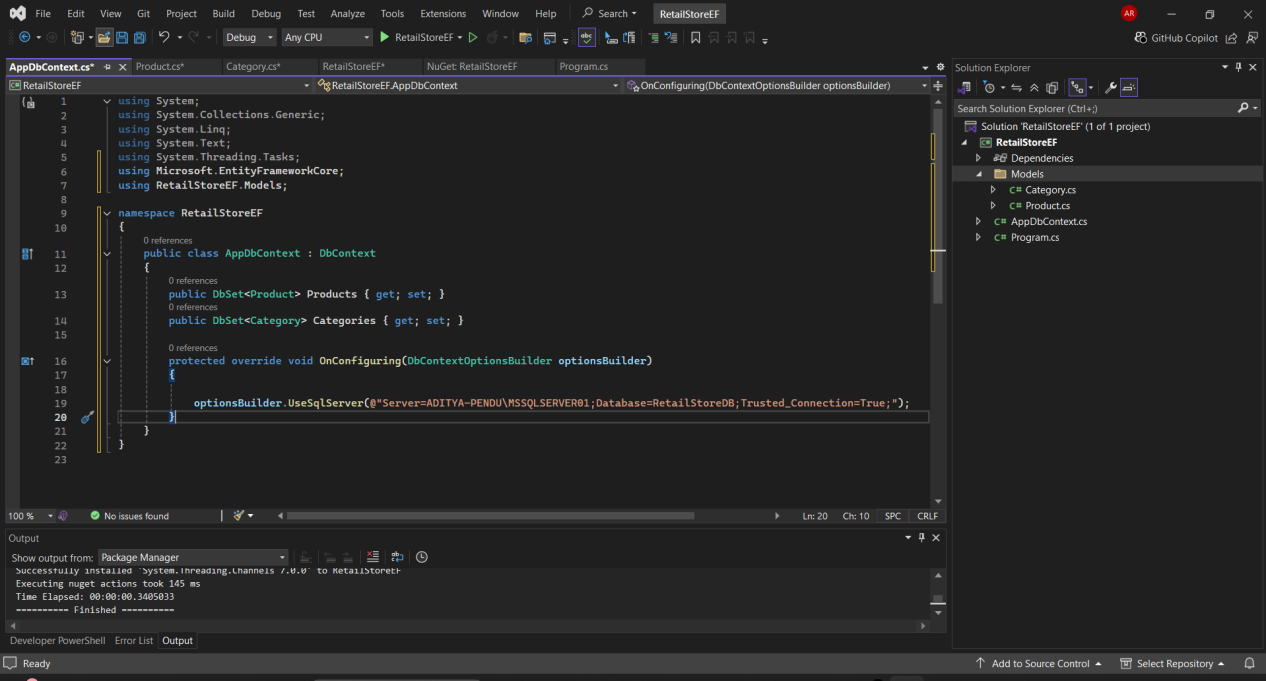
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuild

er) {

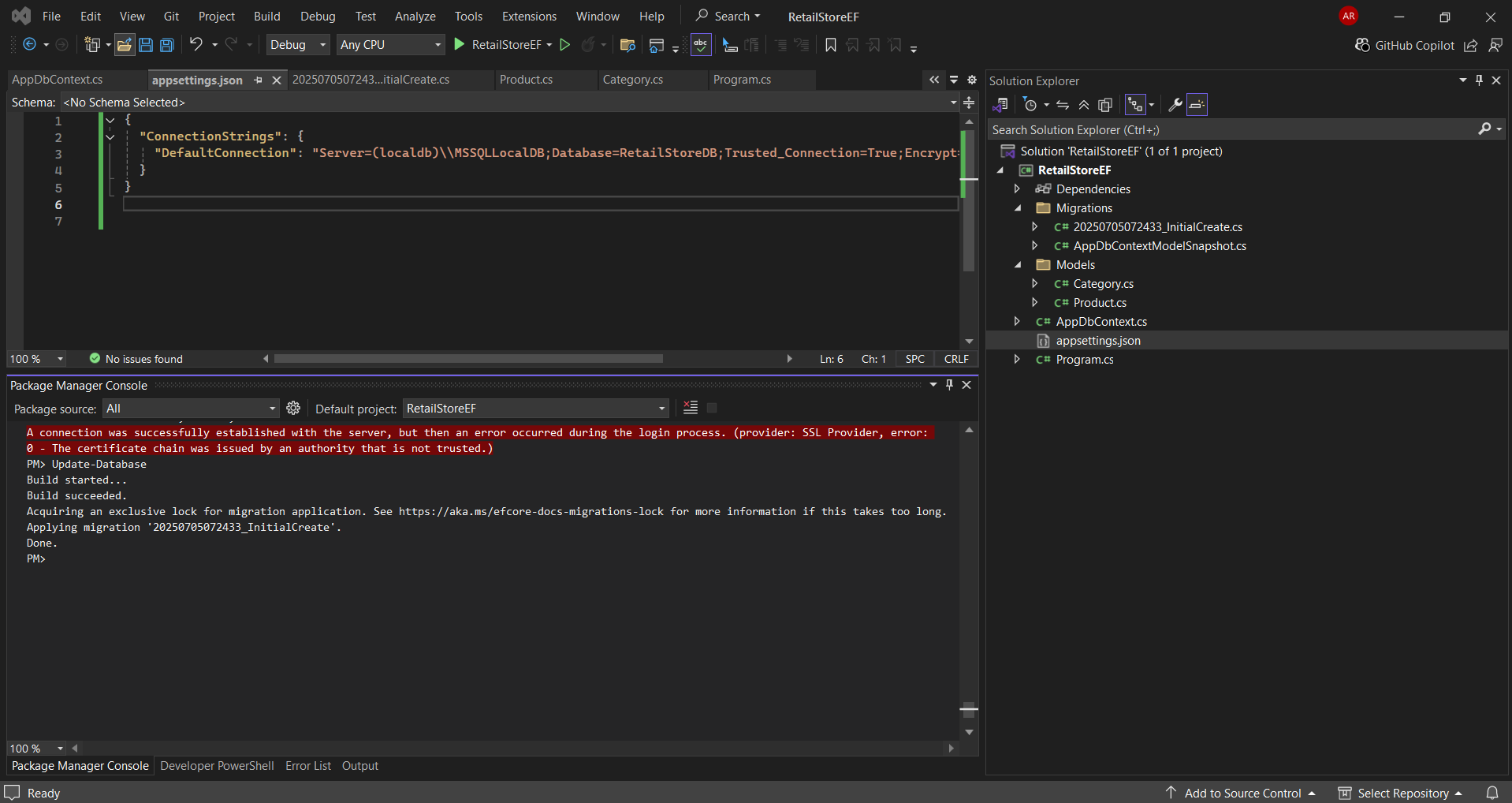
optionsBuilder.UseSqlServer("Your\_Connection\_String\_Here");

}

}



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



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.

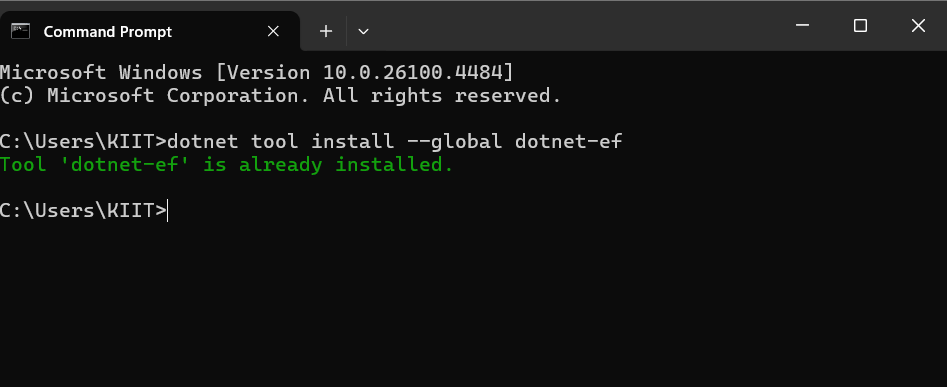
Objective:

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

Steps:

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

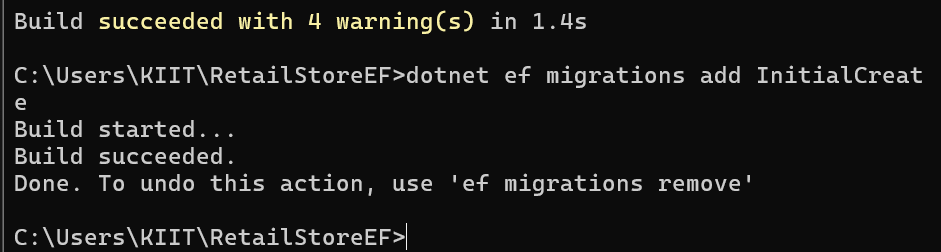
dotnet tool install --global dotnet-ef



2. Create Initial Migration:

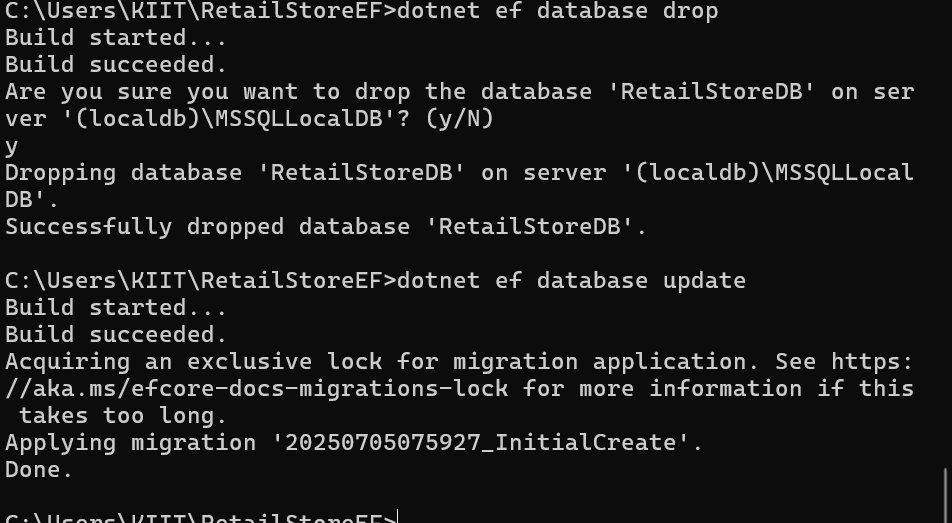
dotnet ef migrations add InitialCreate

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



3. Apply Migration to Create Database:

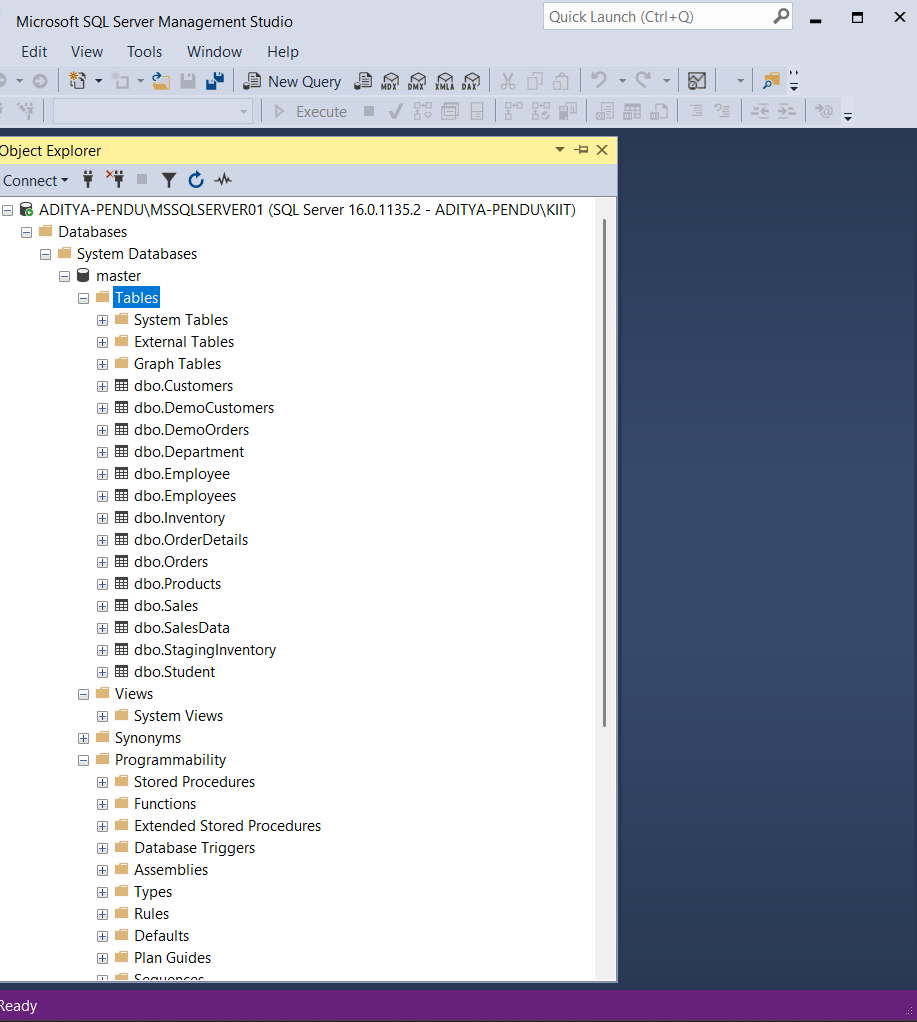
dotnet ef database update



4. Verify in SQL Server:

Open SQL Server Management Studio (SSMS) or Azure Data Studio and confirm

that tables Products and Categories are created.



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

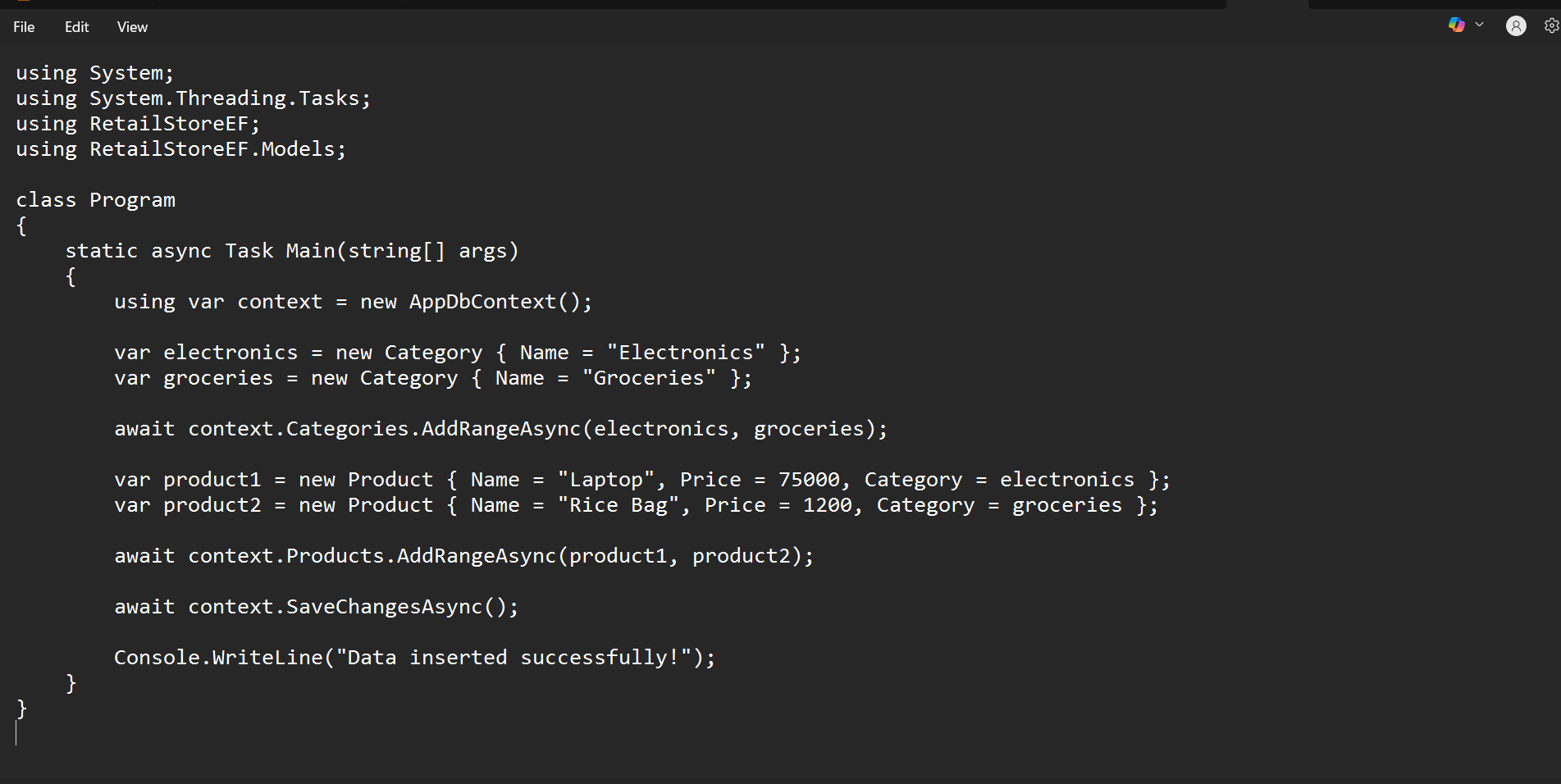
nics };

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

es };

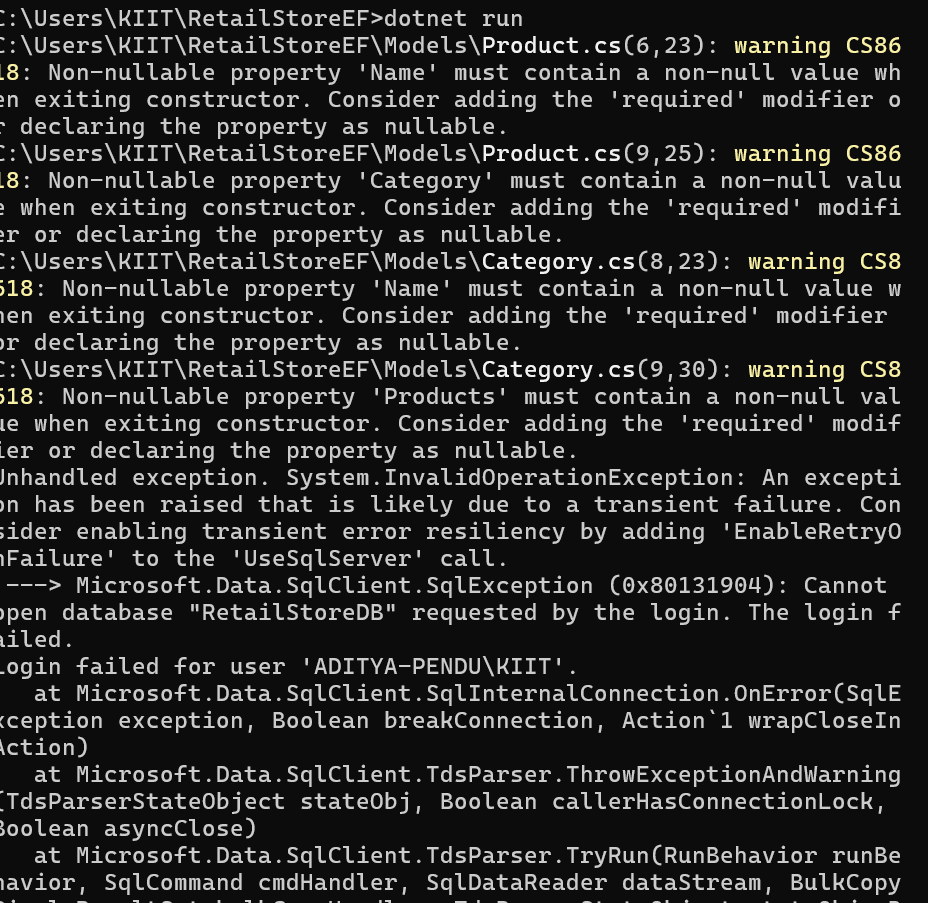
await context.Products.AddRangeAsync(product1, product2);

await context.SaveChangesAsync();



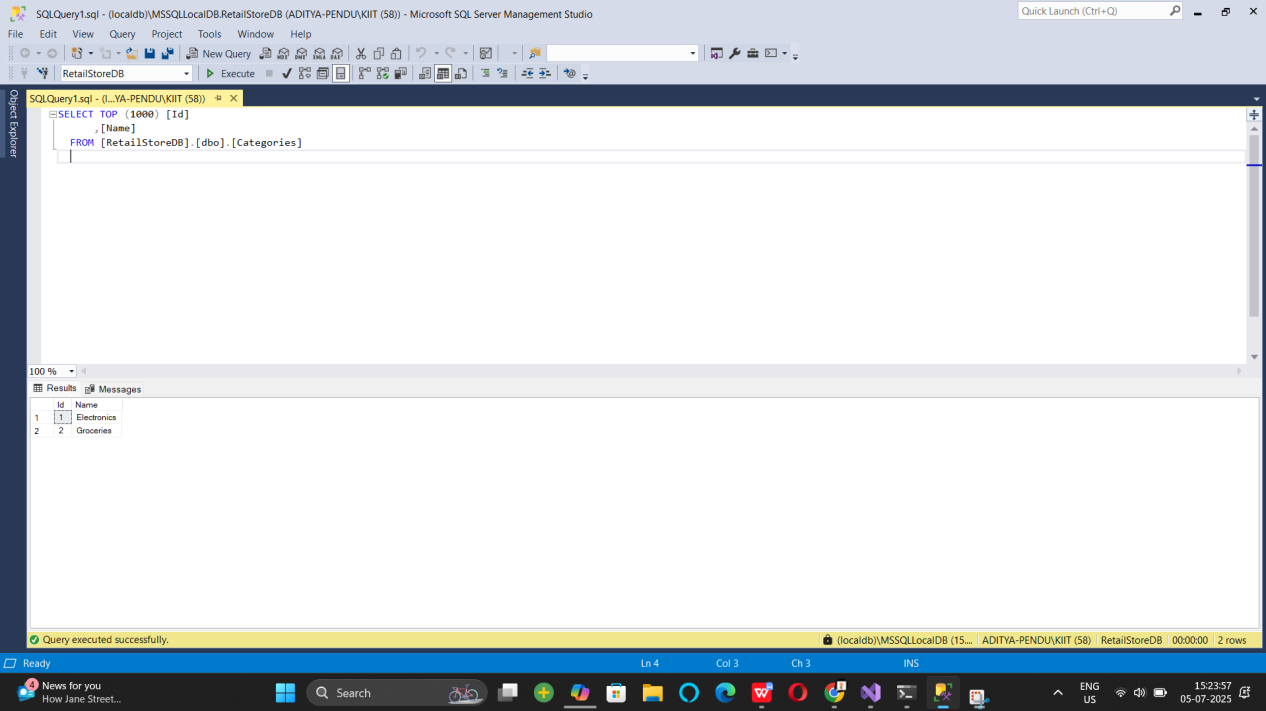
2. Run the App:

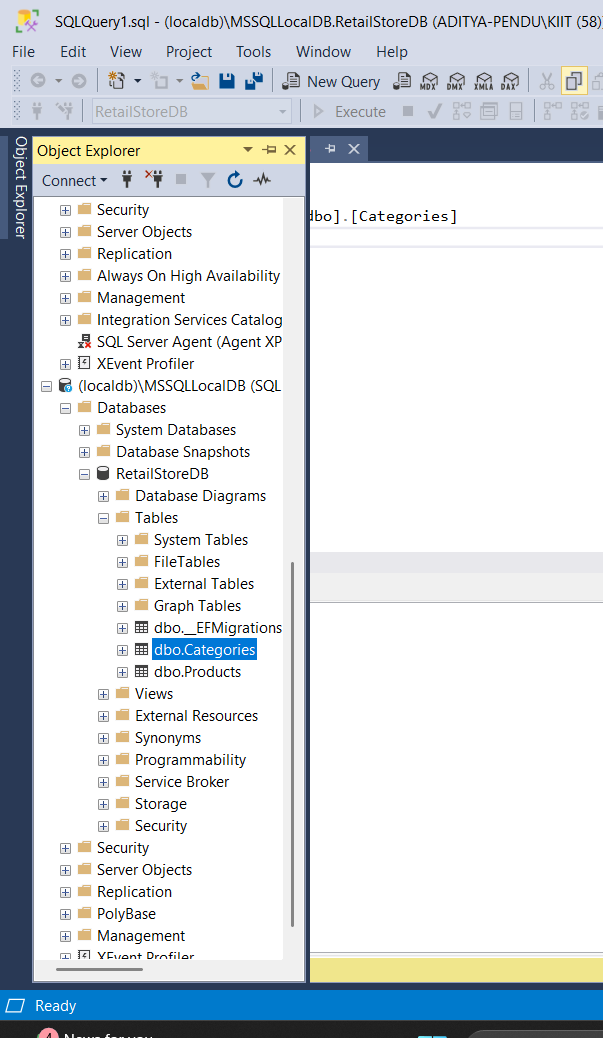
dotnet run



3. Verify in SQL Server:

Check that the data is inserted correctly





Lab 5:

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}");

using System;

using System.Threading.Tasks;

using System.Linq;

using Microsoft.EntityFrameworkCore;

using RetailStoreEF;

using RetailStoreEF.Models;

class Program

{

static async Task Main(string[] args)

{

using var context = new AppDbContext();

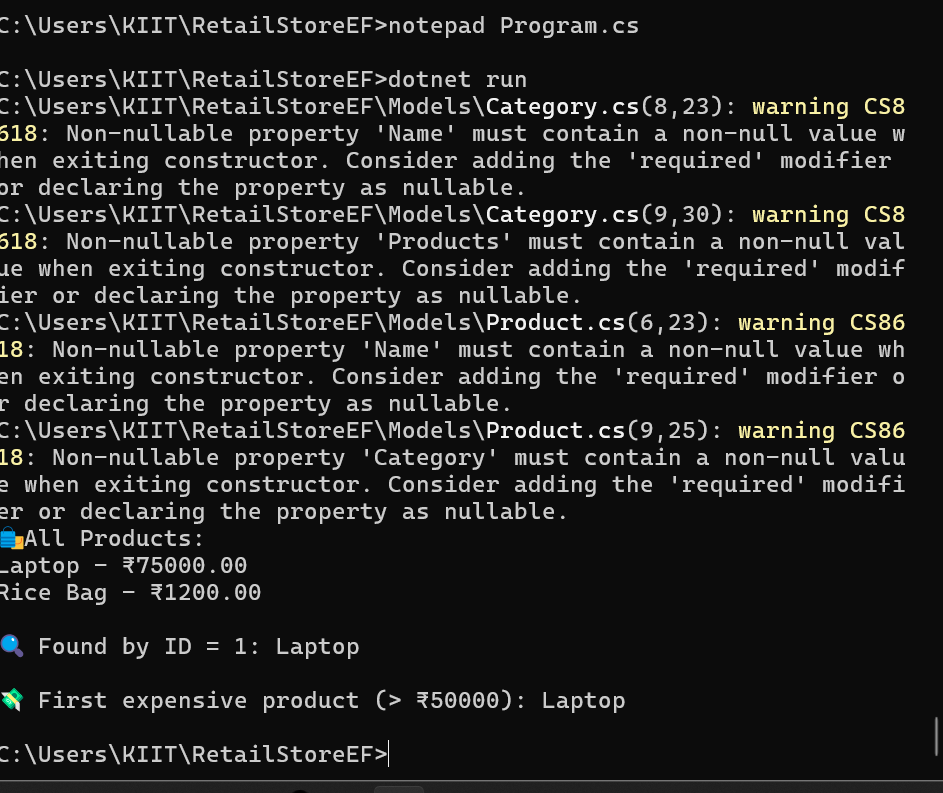
// 1. Retrieve all products

var products = await context.Products.ToListAsync();

Console.WriteLine("🛍 All Products:");

foreach (var p in products)

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



2. Find by ID:

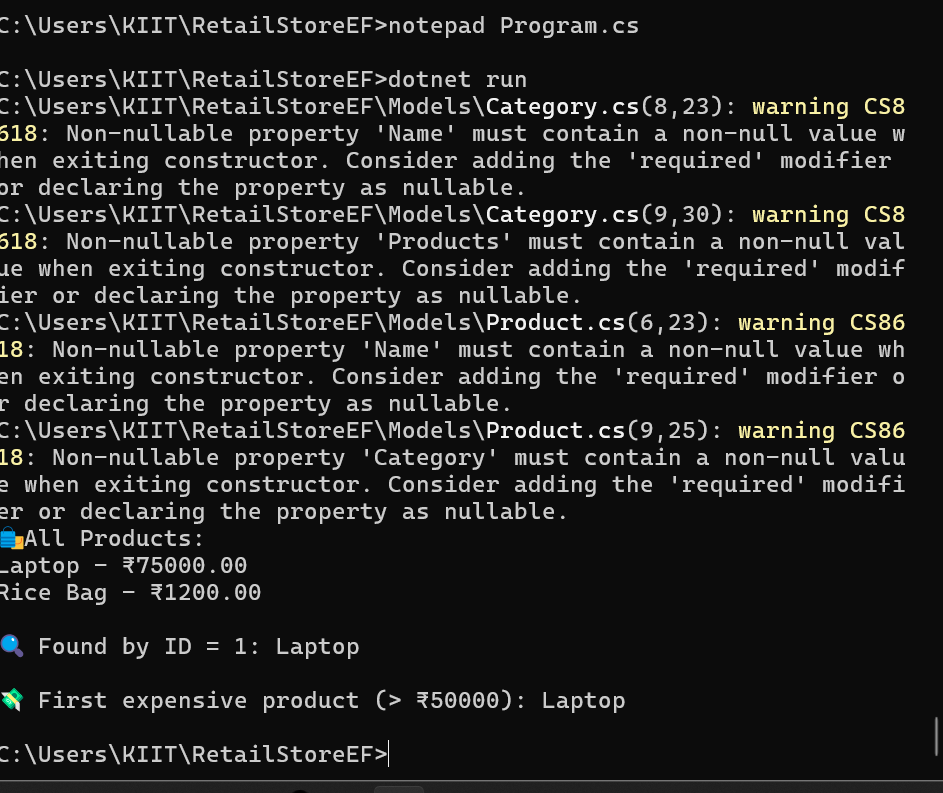
var product = await context.Products.FindAsync(1);

Console.WriteLine($"Found: {product?.Name}");

// 2. Find by ID

var product = await context.Products.FindAsync(1);

Console.WriteLine($"\ Found by ID = 1: {product?.Name}");



3. FirstOrDefault with Condition:

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

0);

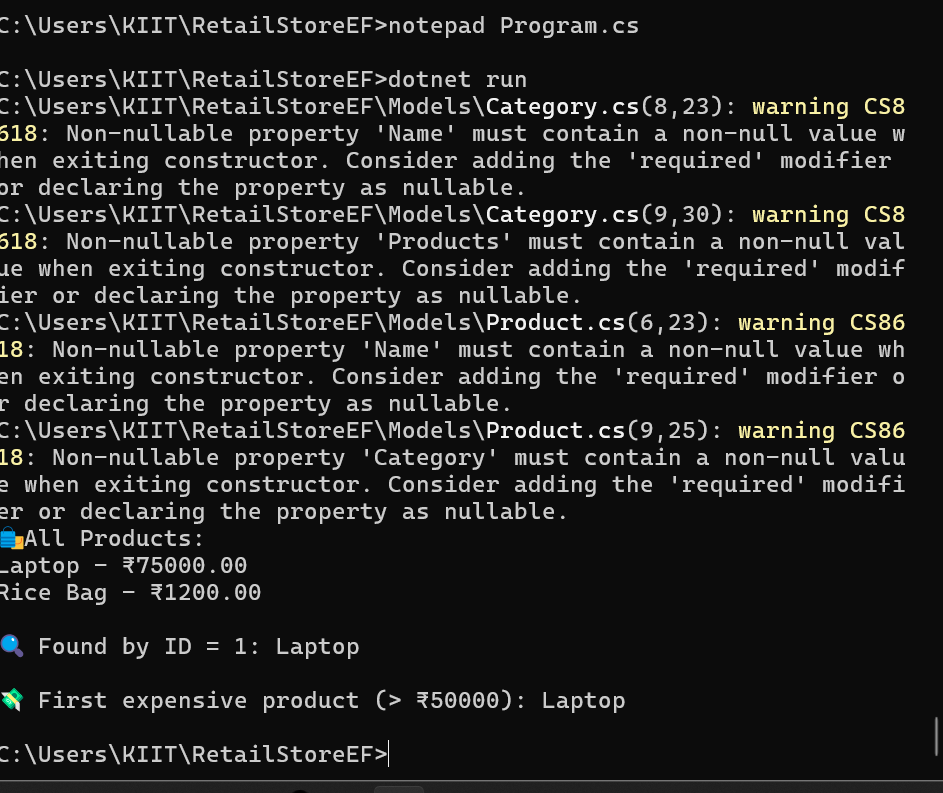
Console.WriteLine($"Expensive: {expensive?.Name}")

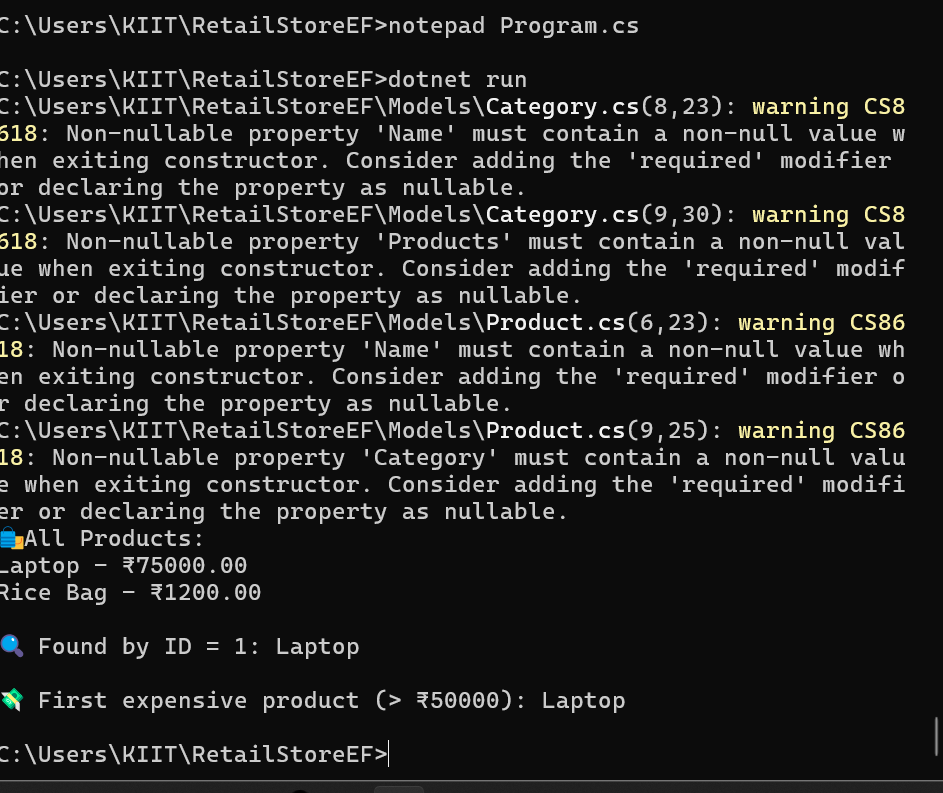
// 3. FirstOrDefault with condition

var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 50000);

Console.WriteLine($"\n💸 First expensive product (> ₹50000): {expensive?.Name}");

}





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

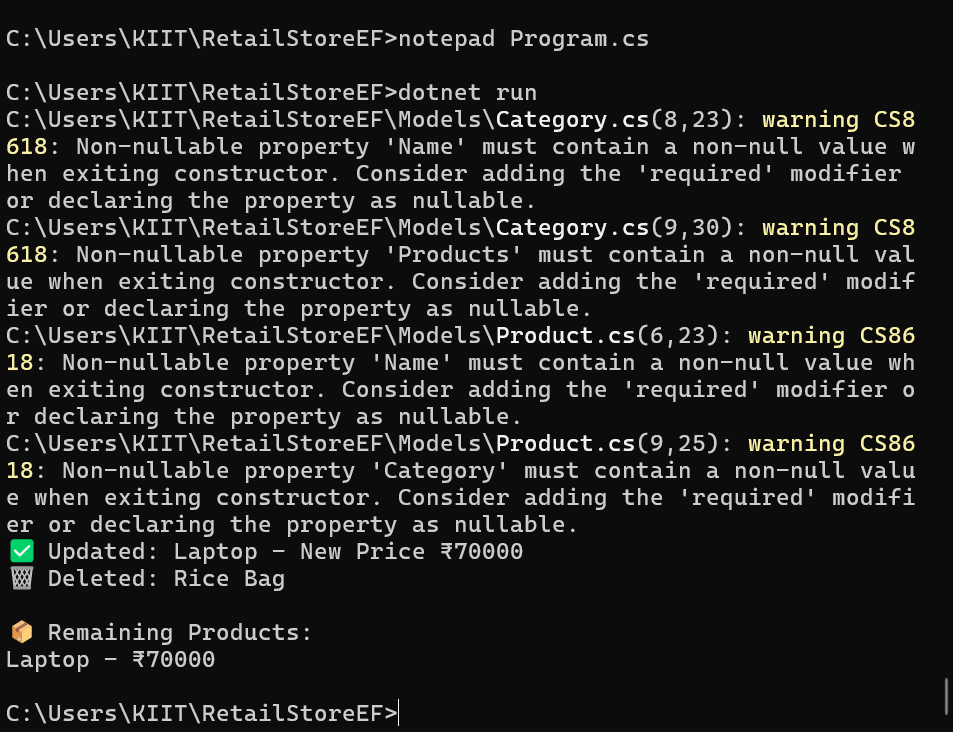
op");

if (product != null) {

product.Price = 70000;

await context.SaveChangesAsync();

}



2. Delete a Product:

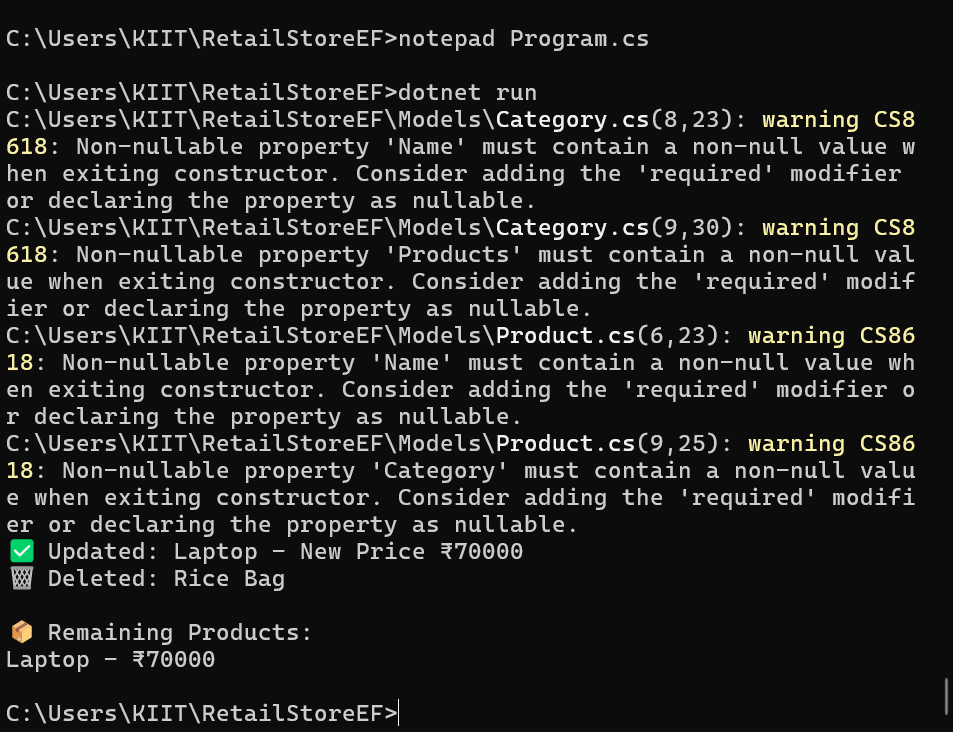
var toDelete = await context.Products.FirstOrDefaultAsync(p => p.Name == "Rice

Bag");

if (toDelete != null) {

context.Products.Remove(toDelete);

await



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()

Ans:

using System;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using RetailStoreEF;

using RetailStoreEF.Models;

class Program

{

static async Task Main(string[] args)

{

using var context = new AppDbContext();

// FILTER and SORT: Products with Price > 1000, sorted descending

var filtered = await context.Products

.Where(p => p.Price > 1000)

.OrderByDescending(p => p.Price)

.ToListAsync();

Console.WriteLine(" Filtered & Sorted Products (Price > ₹1000):");

foreach (var p in filtered)

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

// PROJECT INTO DTO: Only Name and Price

var productDTOs = await context.Products

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

.ToListAsync();

Console.WriteLine("\n🧾 Product DTOs:");

foreach (var dto in productDTOs)

Console.WriteLine($"Name: {dto.Name}, Price: ₹{dto.Price}");

}

}

