**Entity Framework Core 8.0**

**NAME: TANMAY BIMAL**

**SUPERSET ID: 6357868**

**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.

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

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.

3. **EF Core 8.0 Features:**

• JSON column mapping.

• Improved performance with compiled models.

• Interceptors and better bulk operations.

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

**Answer:**

1. ORM (Object-Relational Mapping) is a technique that lets you interact with a relational database using object-oriented programming (like C#), instead of writing raw SQL queries.

How ORM maps C# classes to DB tables:

A C# class becomes a database table.

Each property of the class becomes a column.

Each instance of the class becomes a row in the table.

Example:

public class Product

{

public int ProductId { get; set; } // Becomes primary key column

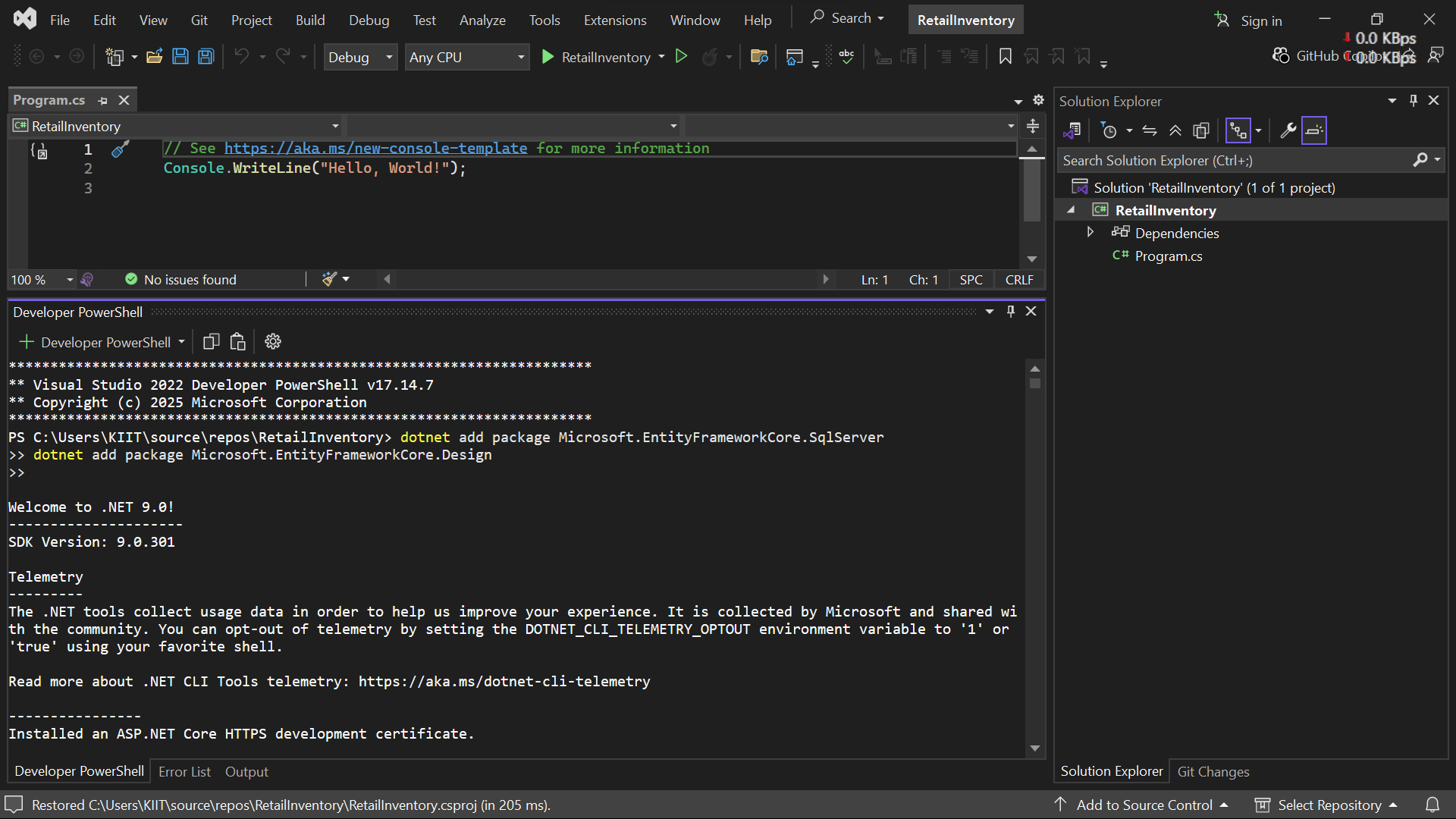
public string Name { get; set; } // Maps to "Name" column

public int Stock { get; set; } // Maps to "Stock" column

}

**Benefits:**

* **Productivity**: Write less SQL; use C# LINQ instead.
* **Maintainability**: Centralized logic in models, easier refactoring.
* **Abstraction**: Avoids boilerplate SQL for common tasks (CRUD).

4 & 5. 

**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).**

**Answer:**

**1.**

**Category.cs**

using System.Collections.Generic;

using RetailInventory.Models;

namespace RetailInventory.Models

{

public class Category

{

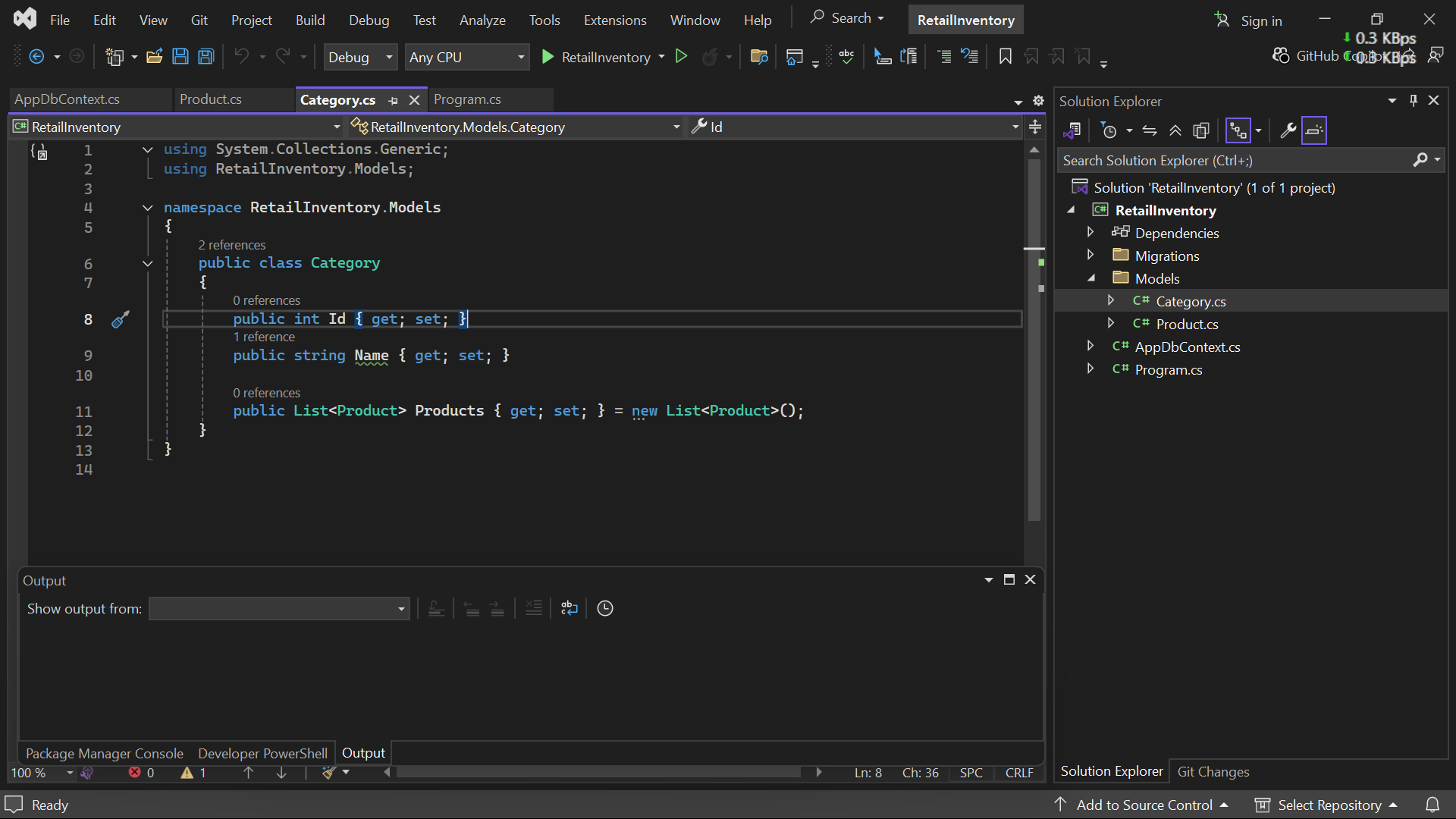
public int Id { get; set; }

public string Name { get; set; }

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

}

}



**Product.cs**

namespace RetailInventory.Models

{

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

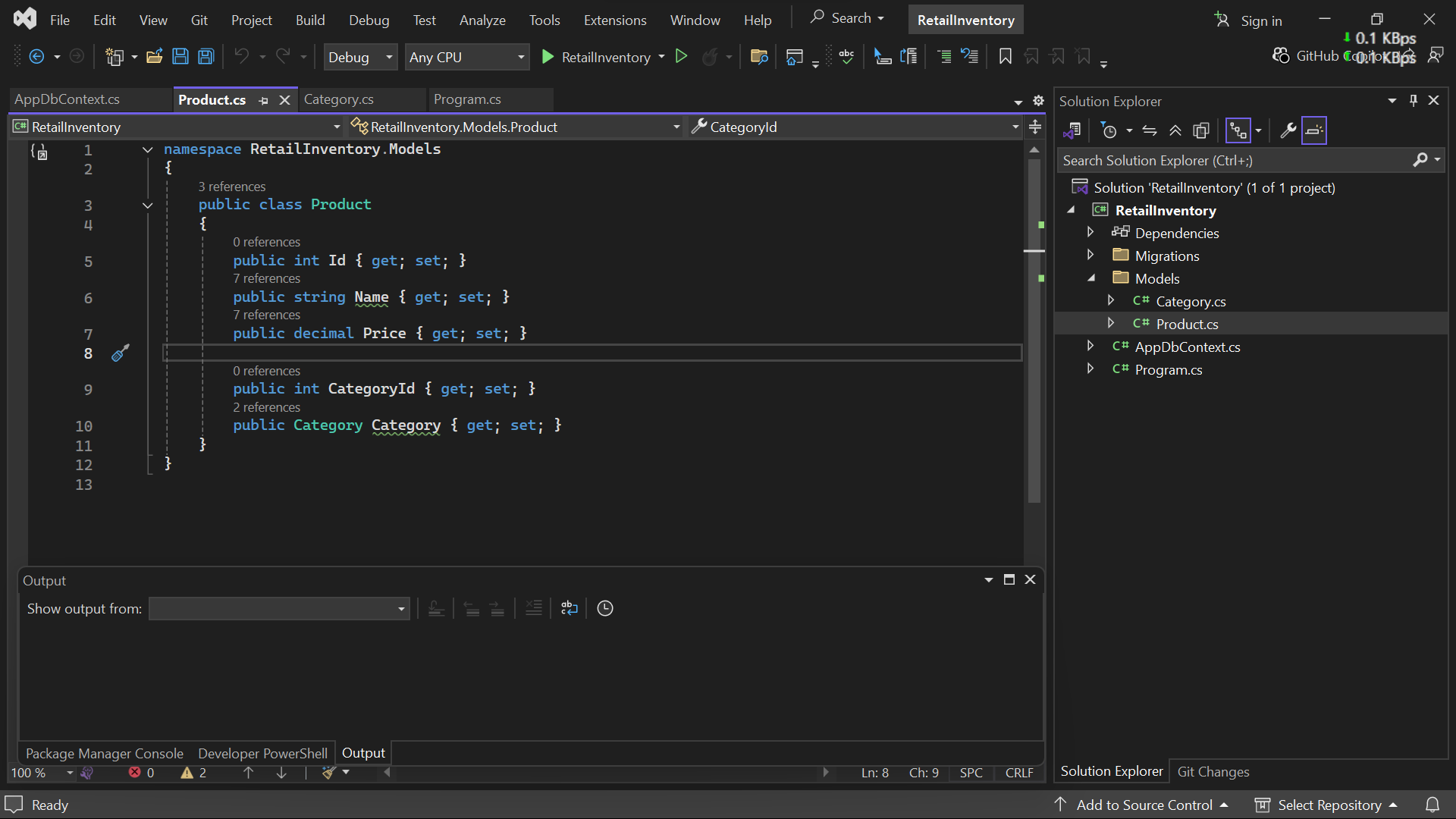
public decimal Price { get; set; }

public int CategoryId { get; set; }

public Category Category { get; set; }

}

}



2.

**AppDbContext.cs**

using Microsoft.EntityFrameworkCore;

using RetailInventory.Models;

namespace RetailInventory

{

public class AppDbContext : DbContext

{

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

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

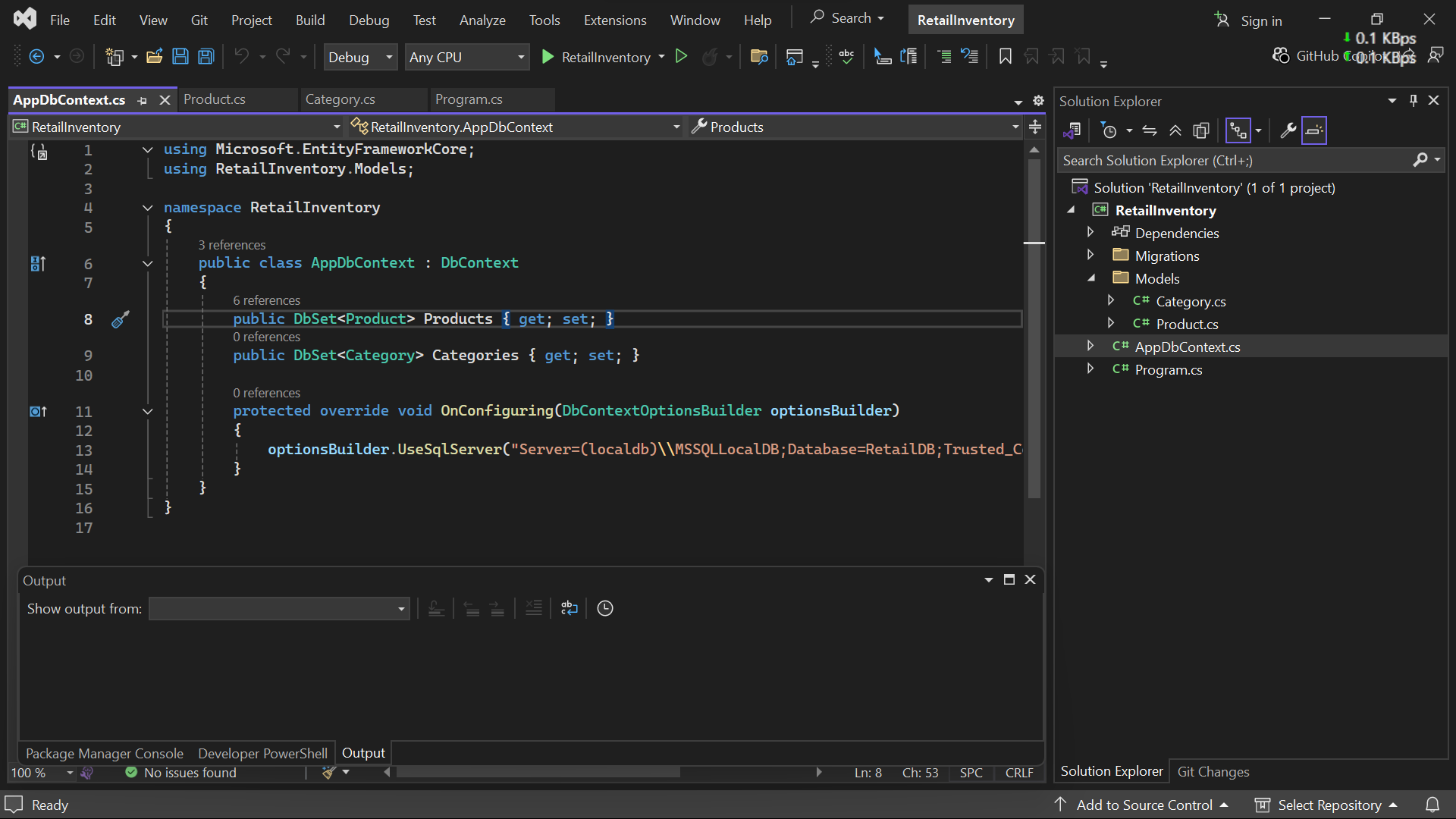
{

optionsBuilder.UseSqlServer("Server=(localdb)\\MSSQLLocalDB;Database=RetailDB;Trusted\_Connection=True;");

}

}

}



**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

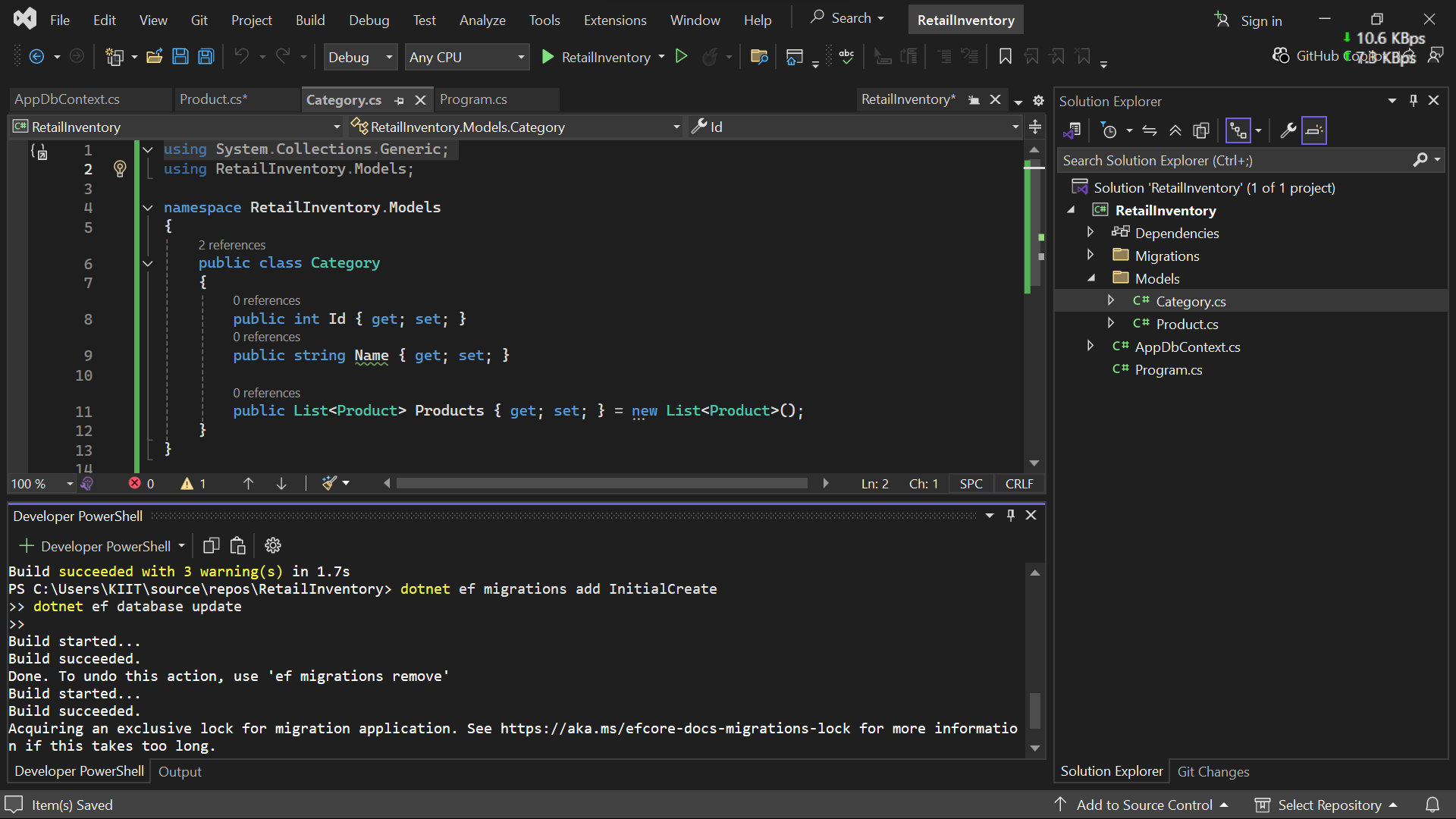
**4. Verify in SQL Server:**

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

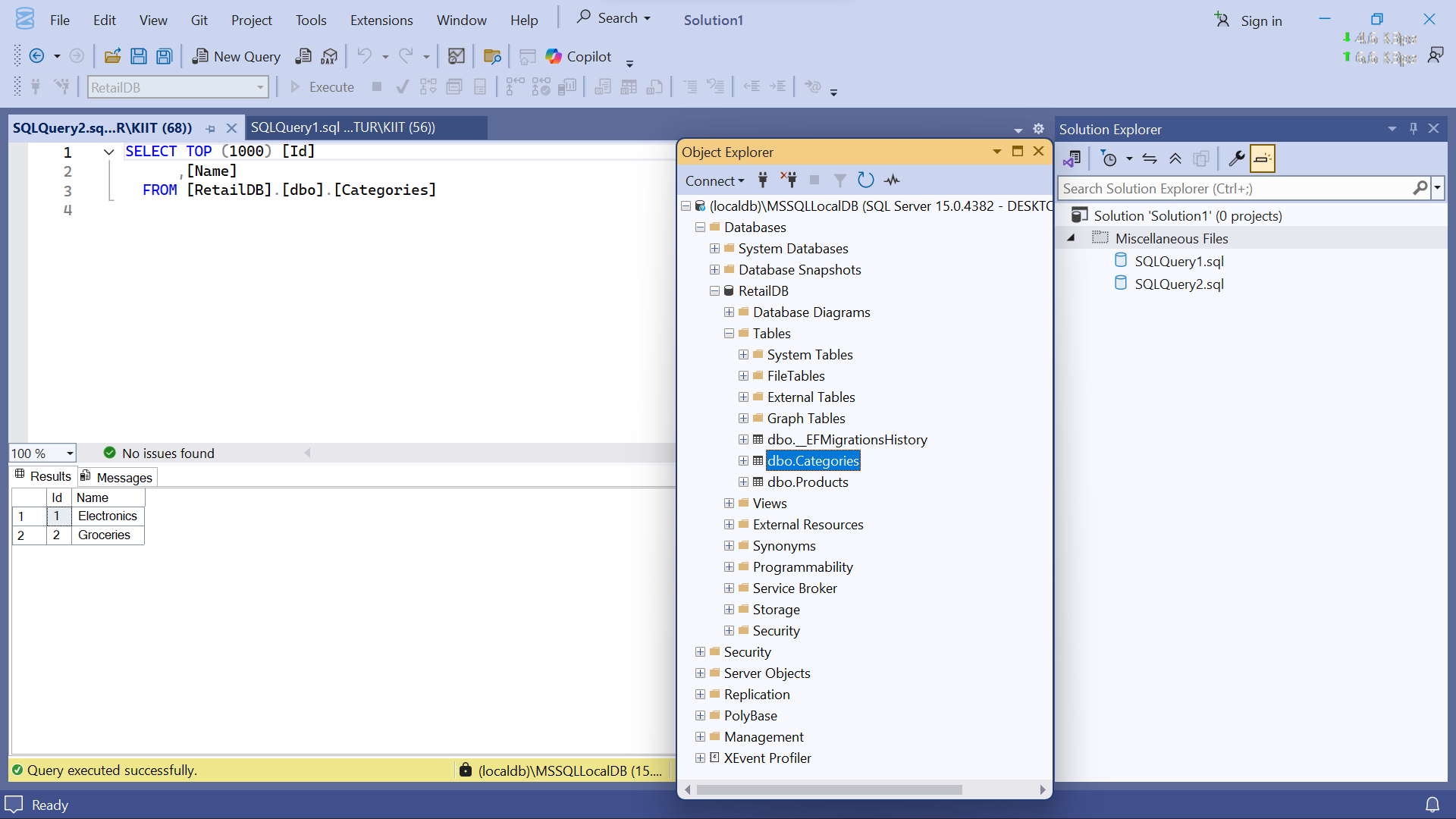
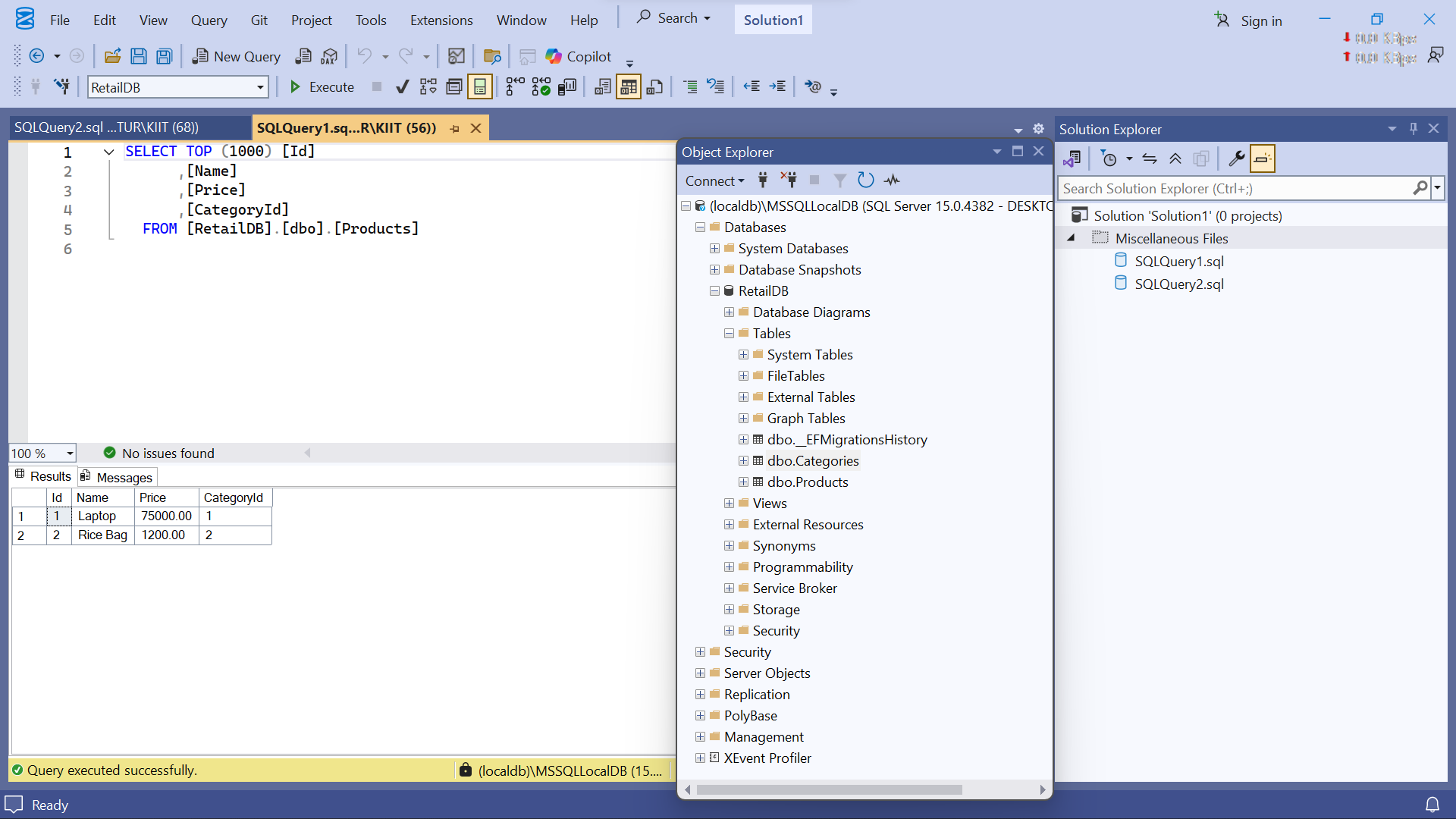
that tables Products and Categories are created.

**Answer:**

1 &2 & 3.



4.



**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.

**Answer:**

1.

**Program.cs**

using RetailInventory;

using RetailInventory.Models;

using System;

using System.Threading.Tasks;

class Program

{

static async Task Main(string[] args)

{

using var context = new AppDbContext();

// Create categories

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

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

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 };

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

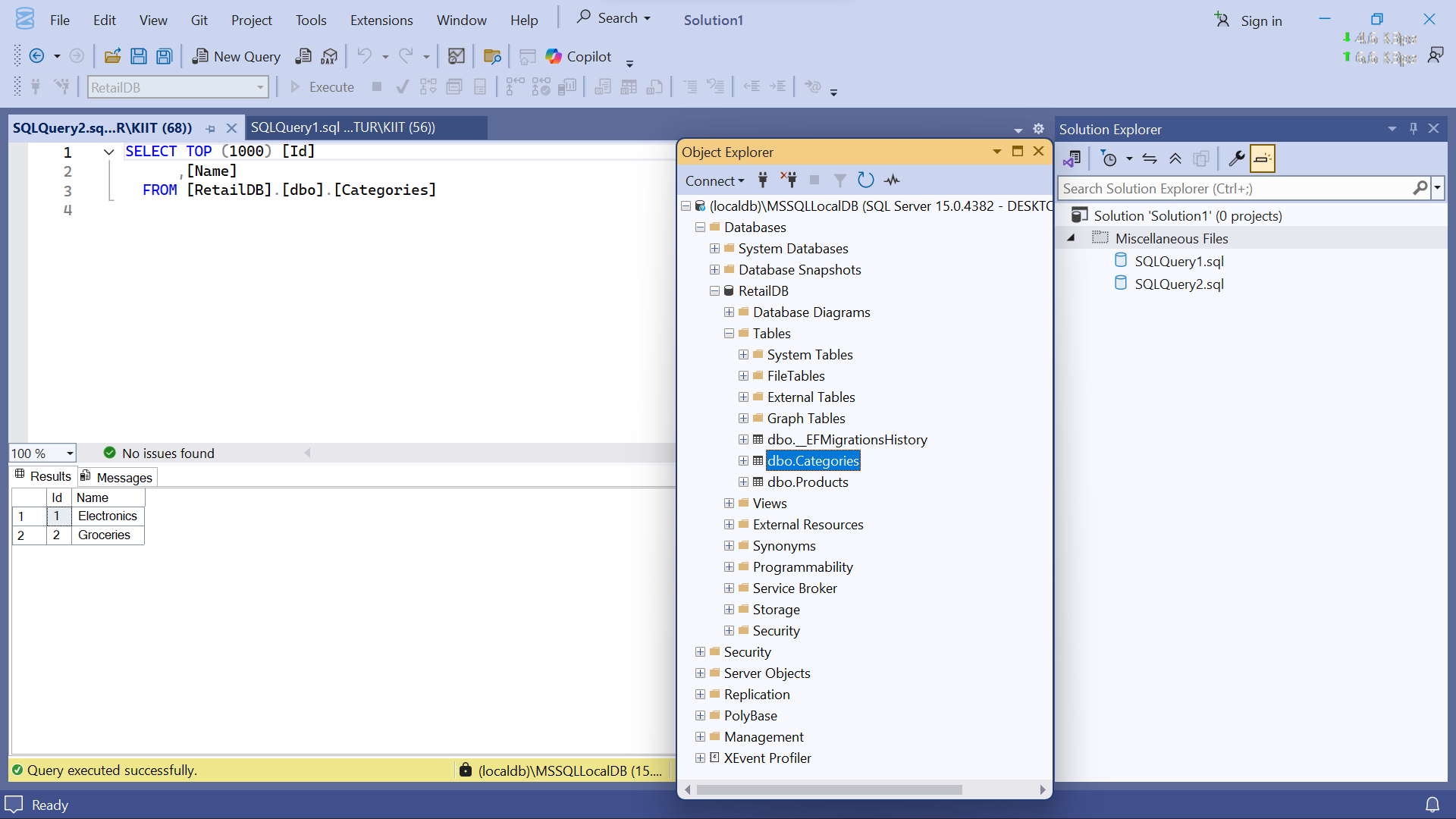
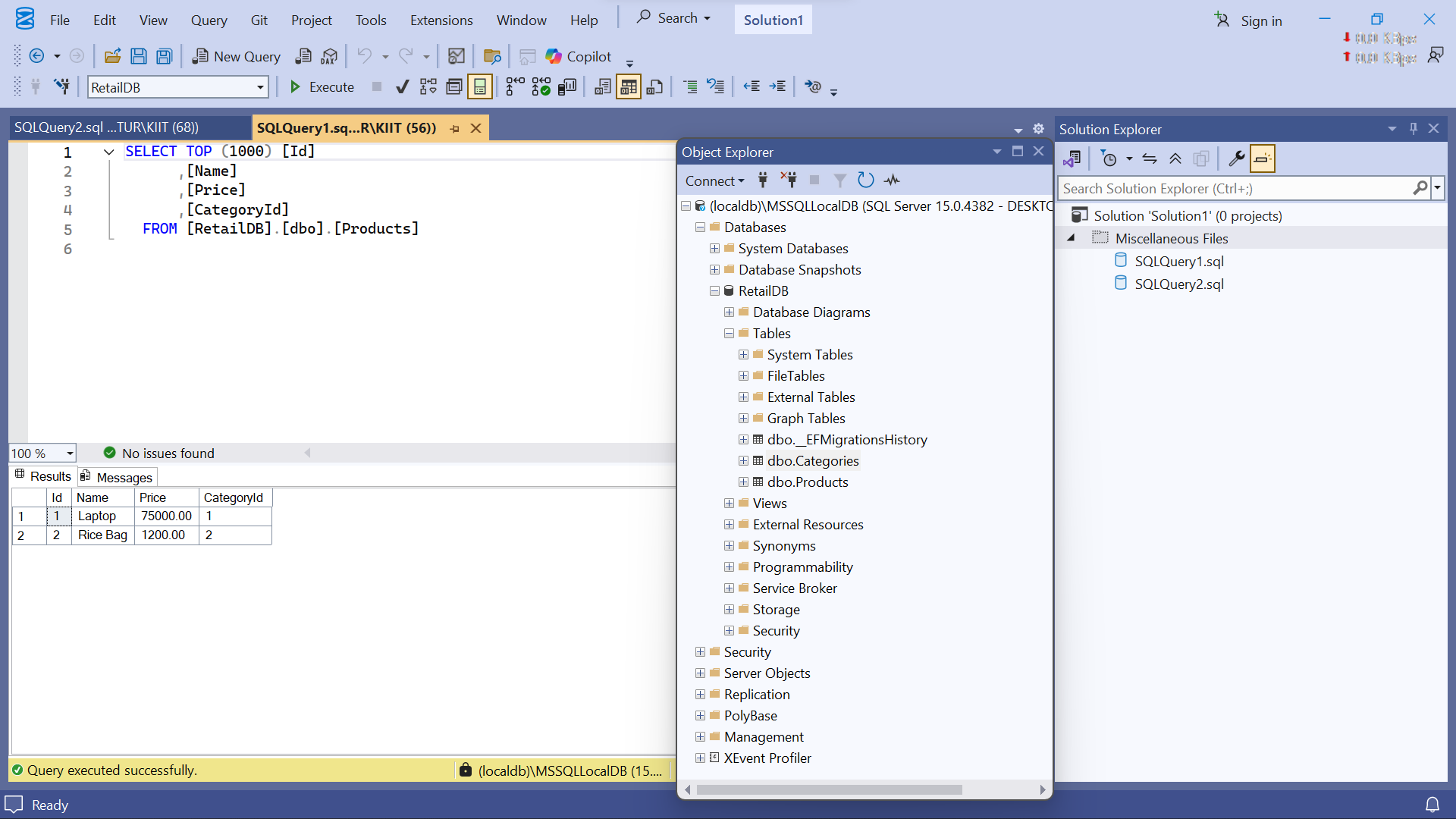
// Save to database

await context.SaveChangesAsync();

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

}

}

2 & 3.

**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}");

**Answer:**

1 & 2 & 3.

**Program.cs**

using RetailInventory;

using RetailInventory.Models;

using Microsoft.EntityFrameworkCore;

using System;

using System.Threading.Tasks;

class Program

{

static async Task Main(string[] args)

{

using var context = new AppDbContext();

Console.WriteLine("=== All Products ===");

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

foreach (var p in products)

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

Console.WriteLine("\n=== Find Product by ID (1) ===");

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

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

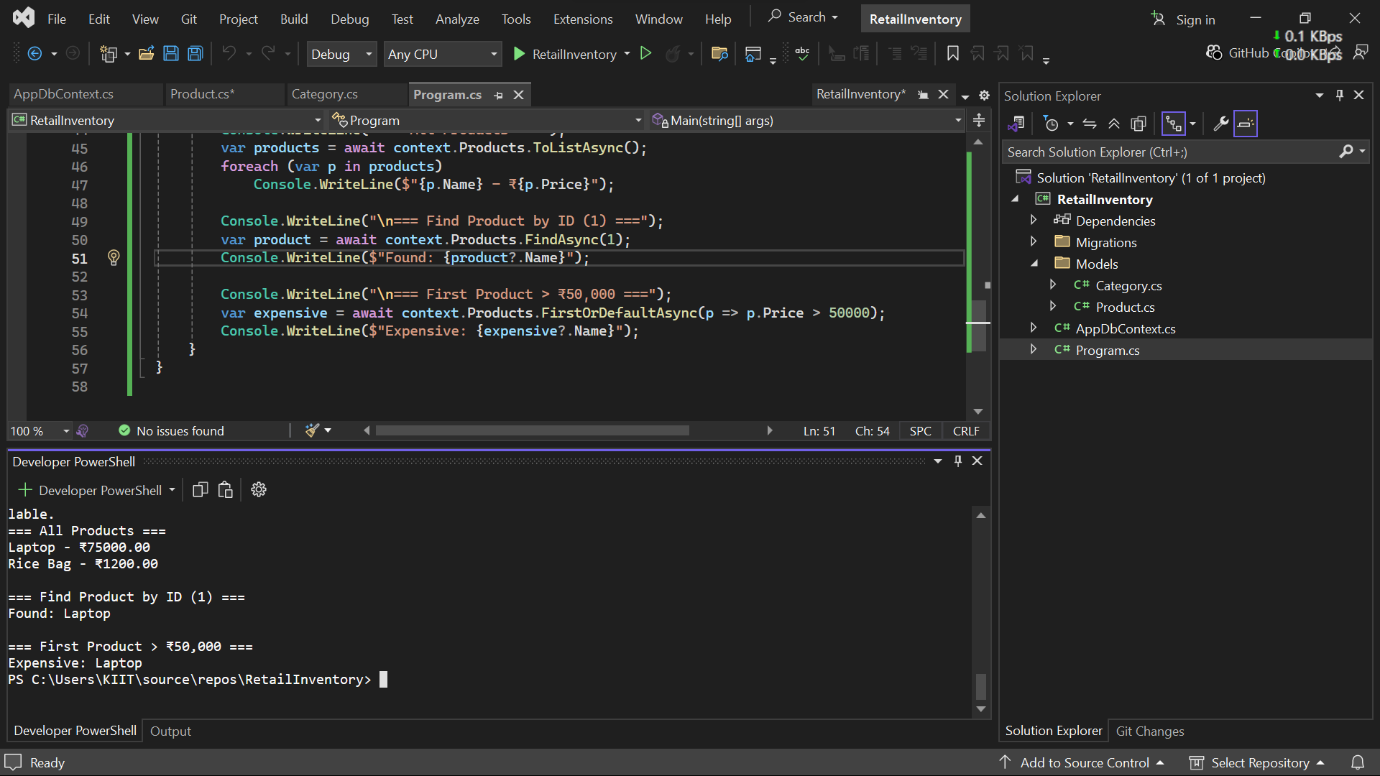
Console.WriteLine("\n=== First Product > ₹50,000 ===");

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

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

}

}



**ADDITIONAL**

**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();

}

**Answer:**

1 & 2.

**Program.cs**

using RetailInventory;

using RetailInventory.Models;

using Microsoft.EntityFrameworkCore;

using System;

using System.Threading.Tasks;

class Program

{

static async Task Main(string[] args)

{

using var context = new AppDbContext();

// Update a Product (Laptop)

var product = await context.Products.FirstOrDefaultAsync(p => p.Name == "Laptop");

if (product != null)

{

product.Price = 70000;

await context.SaveChangesAsync();

Console.WriteLine($"Updated {product.Name} to ₹{product.Price}");

}

//Delete a Product (Rice Bag)

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

if (toDelete != null)

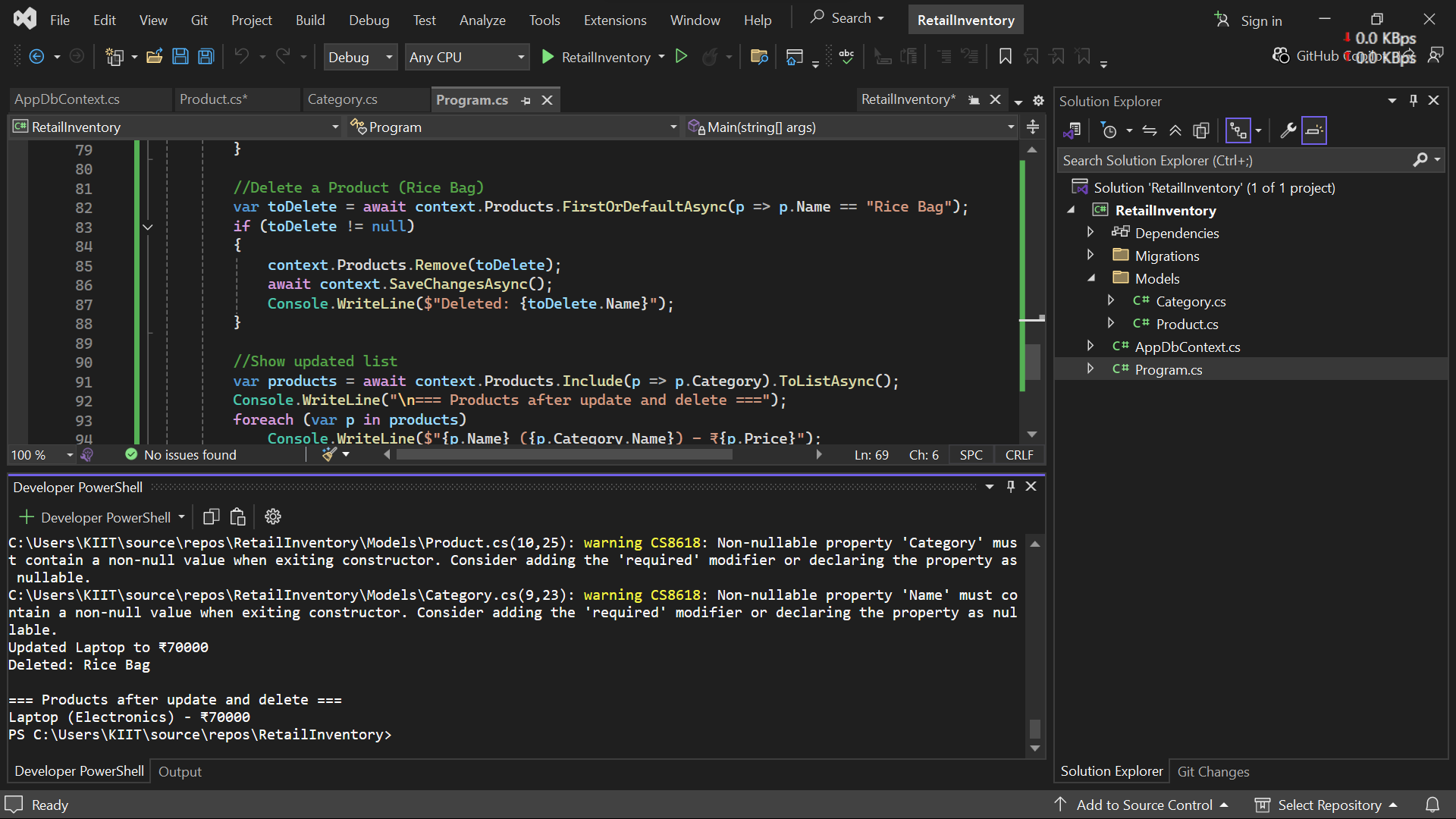
{

context.Products.Remove(toDelete);

await context.SaveChangesAsync();

Console.WriteLine($"Deleted: {toDelete.Name}");

}



**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();

**Answer:**

1 & 2:

**Program.cs**

//Show updated list

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

Console.WriteLine("\n=== Products after update and delete ===");

foreach (var p in products)

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

Console.WriteLine("\n=== Filtered & Sorted Products (Price > ₹1000) ===");

var filtered = await context.Products

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

.OrderByDescending(p => p.Price)

.ToListAsync();

foreach (var p in filtered)

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

Console.WriteLine("\n=== Product DTOs (Name & Price Only) ===");

var productDTOs = await context.Products

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

.ToListAsync();

foreach (var dto in productDTOs)

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

}

}

