**SuperSet ID: 6364957**

**EF Core 8.0 Lab-Wise Implementation Solutions**

**Lab 1: Understanding ORM with a Retail Inventory System**

**Objective**

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

**1. What is ORM?**

Object-Relational Mapping (ORM) is a programming technique that allows us to work with database data using object-oriented programming concepts.

**How ORM maps C# classes to database tables:**

* C# classes correspond to database tables.
* Properties of classes become table columns.
* Objects (instances of classes) represent table rows.
* Relationships between classes are mapped to foreign keys in the database.

**Benefits:**

* **Productivity:** Write less code and focus on business logic.
* **Maintainability:** Changes in one place reflect everywhere.
* **Abstraction:** No need to write complex SQL queries.

1. **EF Core vs EF Framework:**Entity Framework Core (EF Core) and Entity Framework (EF Framework) are both object-relational mappers (ORMs) developed by Microsoft, but they differ significantly in their design and capabilities. EF Core is a modern, lightweight, and cross-platform ORM that supports Windows, Linux, and macOS, making it suitable for a wide range of applications. It is optimized for performance and offers modern features such as LINQ, asynchronous queries, and compiled queries, providing developers with greater flexibility and efficiency. In contrast, EF Framework (also known as Entity Framework 6) is a mature and feature-rich ORM, but it is limited to the Windows platform and is generally heavier, with fewer modern features and less flexibility. While EF Framework is stable and well-established, EF Core is more modular and adaptable to new technologies, making it the preferred choice for new projects that require cross-platform support and modern development practices.

**3. EF Core 8.0 Features**

* **JSON column mapping:** Store JSON data directly in database columns.
* **Improved performance:** Compiled models for faster startup.
* **Interceptors:** Better control over database operations.
* **Bulk operations:** Enhanced performance for batch operations.

**Practical Implementation**

**Step 1: Creating Project**

Create a new console project and navigate into it:

**dotnet new console -n RetailInventory  
cd RetailInventory**

**Step 2: Installing EF Core Packages**

Adding the necessary EF Core packages:

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

Output:

info : PackageReference for package 'Microsoft.EntityFrameworkCore.SqlServer' added  
info : PackageReference for package 'Microsoft.EntityFrameworkCore.Design' added

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

**Objective**

Configure the database context and connect to SQL Server.

**Steps**

1. **Created a Models folder** in your project.
2. **Created the Category model** (Models/Category.cs):

CODE:

using System.ComponentModel.DataAnnotations;

namespace RetailInventory.Models

{

    public class Category

    {

        public int Id { get; set; }

        [Required]

        [StringLength(100)]

        public string Name { get; set; } = string.Empty;

        [StringLength(500)]

        public string? Description { get; set; }

        // Navigation property - One category has many products

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

    }

}

* + The Category class has properties for Id, Name, Description, and a navigation property for related products.

1. **Created the Product model** (Models/Product.cs):

CODE:

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace RetailInventory.Models

{

    public class Product

    {

        public int Id { get; set; }

        [Required]

        [StringLength(200)]

        public string Name { get; set; } = string.Empty;

        [Column(TypeName = "decimal(18,2)")]

        public decimal Price { get; set; }

        public int StockQuantity { get; set; }

        // Foreign key

        public int CategoryId { get; set; }

        // Navigation property - Each product belongs to one category

        public virtual Category Category { get; set; } = null!;

    }

}

* + The Product class includes Id, Name, Price, StockQuantity, CategoryId (foreign key), and a navigation property for its category.

1. **Created a Data folder** in your project.
2. **Created the AppDbContext** (Data/AppDbContext.cs):

CODE:

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)

        {

            // Connection string for LocalDB

            string connectionString = @"Server=(localdb)\mssqllocaldb;Database=RetailInventoryDB;Trusted\_Connection=true;MultipleActiveResultSets=true;";

            optionsBuilder.UseSqlServer(connectionString);

        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)

        {

            // Configure relationships and constraints

            modelBuilder.Entity<Product>()

                .HasOne(p => p.Category)

                .WithMany(c => c.Products)

                .HasForeignKey(p => p.CategoryId);

            // Add indexes for better performance

            modelBuilder.Entity<Category>()

                .HasIndex(c => c.Name)

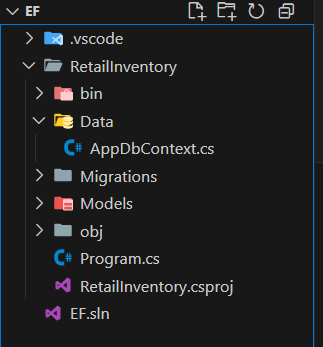
                .IsUnique();

        }

    }

}

* + The AppDbContext class inherits from DbContext and defines DbSet<Product> and DbSet<Category>.
  + The OnConfiguring method sets up the SQL Server connection string.
  + The OnModelCreating method configures relationships and adds a unique index on category names.

**Project structure:**

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

**Objective**

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

1. **Creating the initial migration UsinG:**

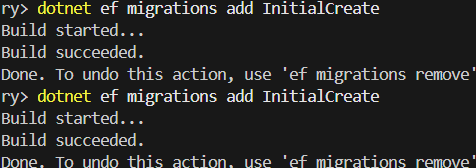
**dotnet ef migrations add InitialCreate**

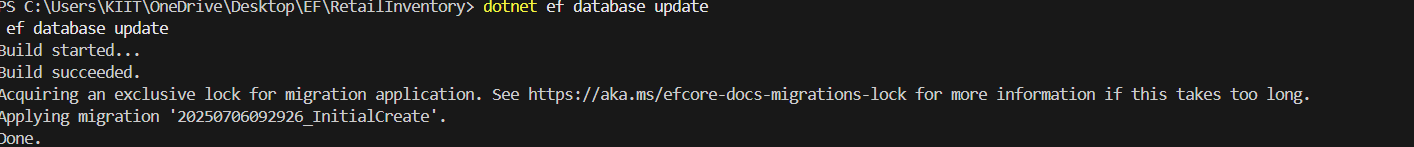
* + This creates a Migrations folder with migration files.

1. **Apply the migration to create the database:**

**dotnet ef database update**

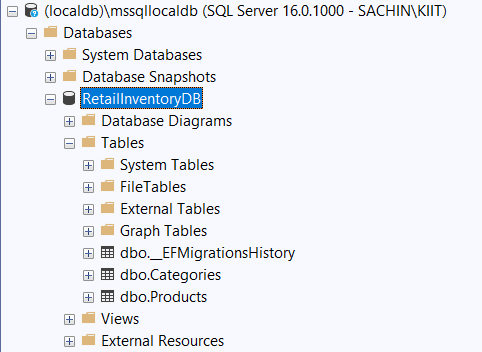
**Result**





1. **Verify database creation:**
   * Use SQL Server Management Studio to check for the RetailInventoryDB database and ensure the Products and Categories tables exist.

**Result of database schema:**



**Lab 4: Inserting Initial Data into the Database**

**Objective**

Use EF Core to insert records using AddAsync and SaveChangesAsync.

**Steps**

1. **Updating Program.cs for data insertion:**

**Code:**using Microsoft.EntityFrameworkCore;

using RetailInventory.Data;

using RetailInventory.Models;

namespace RetailInventory

{

    class Program

    {

        static async Task Main(string[] args)

        {

            Console.WriteLine("=== Retail Inventory System ===");

            Console.WriteLine("Lab 4: Inserting Initial Data");

            using var context = new AppDbContext();

            // Ensure database exists

            await context.Database.EnsureCreatedAsync();

            // Check if data already exists

            var existingCategories = await context.Categories.CountAsync();

            if (existingCategories > 0)

            {

                Console.WriteLine("Data already exists in database.");

                return;

            }

            Console.WriteLine("Inserting initial data...");

            // Create categories

            var electronics = new Category

            {

                Name = "Electronics",

                Description = "Electronic devices and accessories"

            };

            var groceries = new Category

            {

                Name = "Groceries",

                Description = "Food and household items"

            };

            // Add categories to context

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

            // Create products

            var product1 = new Product

            {

                Name = "Laptop",

                Price = 75000,

                StockQuantity = 10,

                Category = electronics

            };

            var product2 = new Product

            {

                Name = "Rice Bag",

                Price = 1200,

                StockQuantity = 50,

                Category = groceries

            };

            var product3 = new Product

            {

                Name = "Wireless Mouse",

                Price = 2500,

                StockQuantity = 25,

                Category = electronics

            };

            var product4 = new Product

            {

                Name = "Organic Honey",

                Price = 450,

                StockQuantity = 30,

                Category = groceries

            };

            // Add products to context

            await context.Products.AddRangeAsync(product1, product2, product3, product4);

            // Save all changes to database

            await context.SaveChangesAsync();

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

            Console.WriteLine("\nInserted Categories:");

            Console.WriteLine($"- {electronics.Name} (ID: {electronics.Id})");

            Console.WriteLine($"- {groceries.Name} (ID: {groceries.Id})");

            Console.WriteLine("\nInserted Products:");

            Console.WriteLine($"- {product1.Name} - ₹{product1.Price} (ID: {product1.Id})");

            Console.WriteLine($"- {product2.Name} - ₹{product2.Price} (ID: {product2.Id})");

            Console.WriteLine($"- {product3.Name} - ₹{product3.Price} (ID: {product3.Id})");

            Console.WriteLine($"- {product4.Name} - ₹{product4.Price} (ID: {product4.Id})");

        }

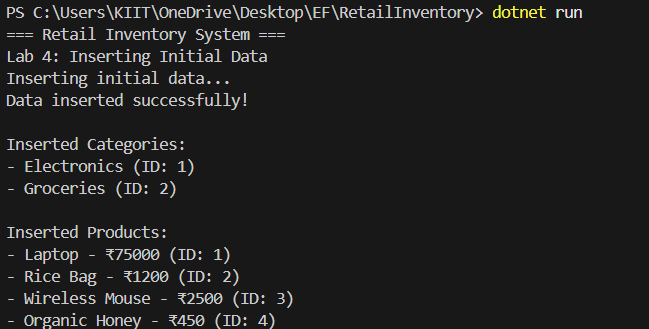
    }

}

1. **Runing the application:**

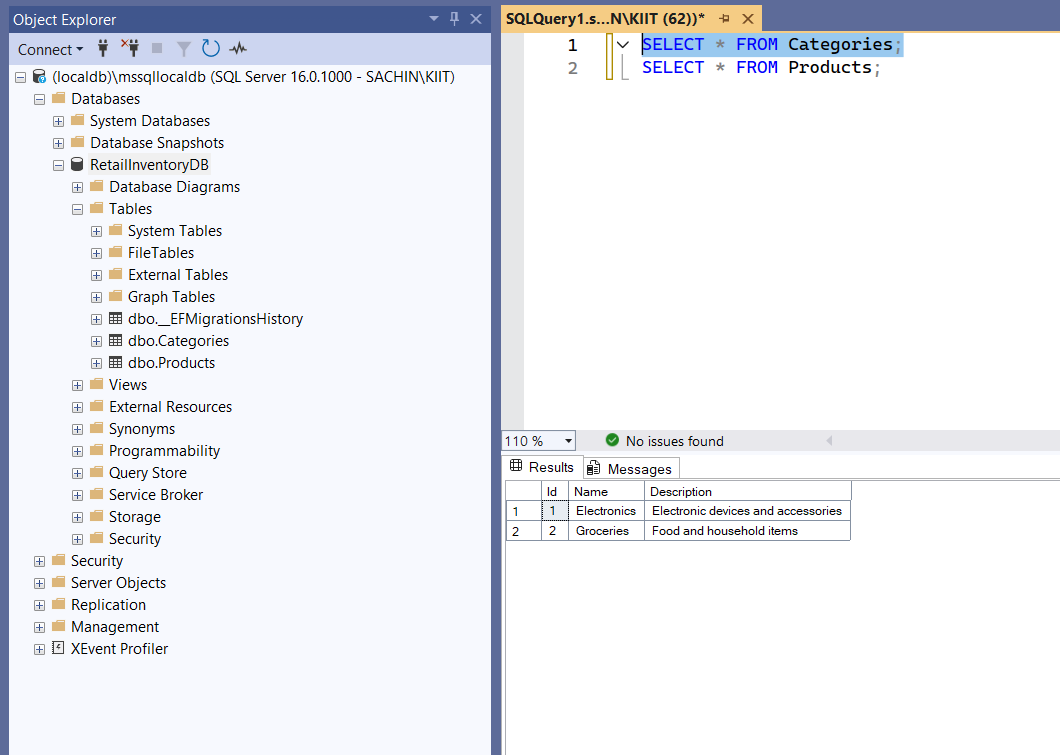
**dotnet run**

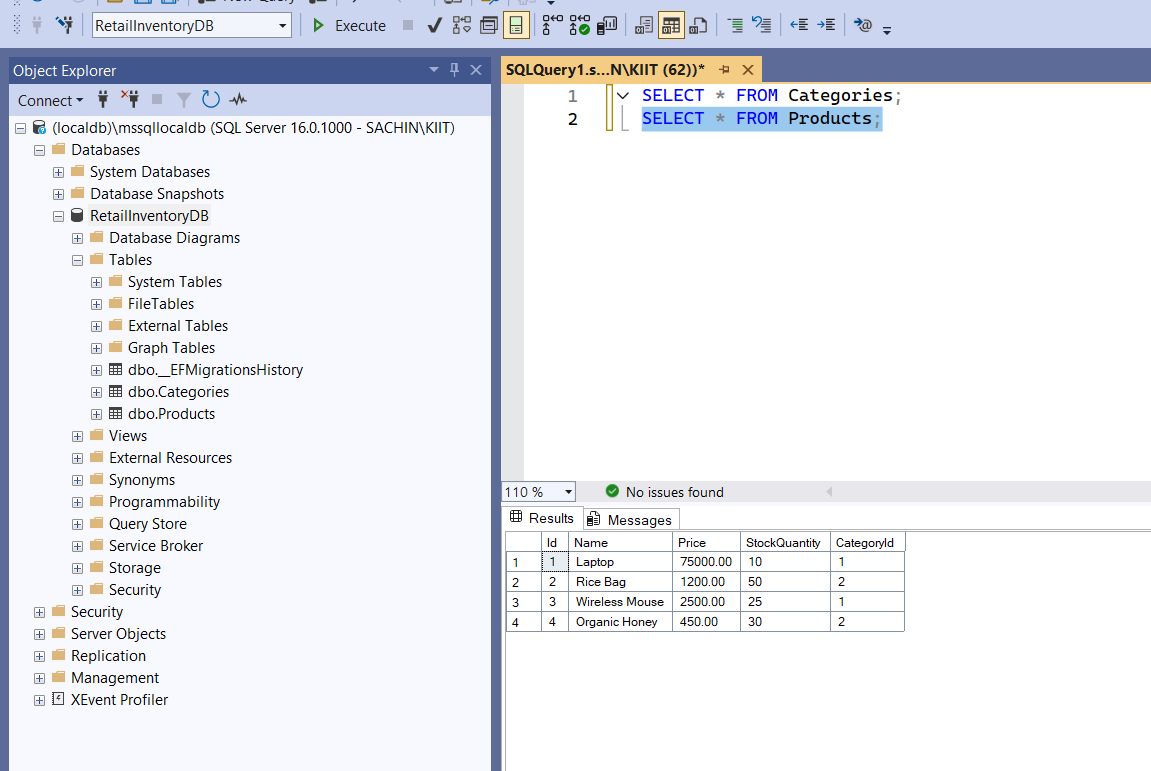
**Output:**



1. **Verify in SQL Server:**
   * Use SQL queries to check the contents of the Categories and Products tables.

**Output**:





**Lab 5: Retrieving Data from the Database**

**Objective**

Use Find, FirstOrDefault, and ToListAsync to retrieve data.

**Steps**

1. **Updating Program.cs for data retrieval:**
   * Ensure the database and data exist.
   * Retrieve all products and display them.
   * Find a product by ID.
   * Find expensive products (e.g., price above ₹50,000).
   * Find products by category (e.g., Electronics).
   * Display categories with product counts.

**CODE:**

using Microsoft.EntityFrameworkCore;

using RetailInventory.Data;

using RetailInventory.Models;

namespace RetailInventory

{

    class Program

    {

        static async Task Main(string[] args)

        {

            Console.WriteLine("=== Retail Inventory System ===");

            Console.WriteLine("Lab 5: Retrieving Data from Database\n");

            using var context = new AppDbContext();

            // Ensure database and data exist

            await context.Database.EnsureCreatedAsync();

            await EnsureDataExistsAsync(context);

            // Lab 5 Operations

            await RetrieveAllProductsAsync(context);

            await FindProductByIdAsync(context);

            await FindExpensiveProductAsync(context);

            await FindProductsByCategoryAsync(context);

            await DisplayCategoryWithProductsAsync(context);

        }

        static async Task EnsureDataExistsAsync(AppDbContext context)

        {

            var count = await context.Products.CountAsync();

            if (count == 0)

            {

                Console.WriteLine("No data found. Inserting sample data...");

                var electronics = new Category { Name = "Electronics", Description = "Electronic devices" };

                var groceries = new Category { Name = "Groceries", Description = "Food items" };

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

                var products = new List<Product>

                {

                    new Product { Name = "Laptop", Price = 75000, StockQuantity = 10, Category = electronics },

                    new Product { Name = "Rice Bag", Price = 1200, StockQuantity = 50, Category = groceries },

                    new Product { Name = "Wireless Mouse", Price = 2500, StockQuantity = 25, Category = electronics },

                    new Product { Name = "Organic Honey", Price = 450, StockQuantity = 30, Category = groceries }

                };

                await context.Products.AddRangeAsync(products);

                await context.SaveChangesAsync();

                Console.WriteLine("Sample data inserted.\n");

            }

        }

        // 1. Retrieve All Products

        static async Task RetrieveAllProductsAsync(AppDbContext context)

        {

            Console.WriteLine("=== 1. RETRIEVE ALL PRODUCTS ===");

            var products = await context.Products

                .Include(p => p.Category)  // Include related category data

                .ToListAsync();

            Console.WriteLine($"Total products found: {products.Count}");

            Console.WriteLine($"{"Name",-15} {"Price",-10} {"Stock",-8} {"Category",-12}");

            Console.WriteLine(new string('-', 50));

            foreach (var p in products)

            {

                Console.WriteLine($"{p.Name,-15} ₹{p.Price,-9} {p.StockQuantity,-8} {p.Category.Name,-12}");

            }

            Console.WriteLine();

        }

        // 2. Find Product by ID

        static async Task FindProductByIdAsync(AppDbContext context)

        {

            Console.WriteLine("=== 2. FIND PRODUCT BY ID ===");

            // Find product with ID 1

            var product = await context.Products

                .Include(p => p.Category)

                .FirstOrDefaultAsync(p => p.Id == 1);

            if (product != null)

            {

                Console.WriteLine($"Product found:");

                Console.WriteLine($"ID: {product.Id}");

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

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

                Console.WriteLine($"Stock: {product.StockQuantity}");

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

            }

            else

            {

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

            }

            Console.WriteLine();

        }

        // 3. Find Expensive Products

        static async Task FindExpensiveProductAsync(AppDbContext context)

        {

            Console.WriteLine("=== 3. FIND EXPENSIVE PRODUCTS ===");

            // Find products above ₹50000

            var expensive = await context.Products

                .Include(p => p.Category)

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

            if (expensive != null)

            {

                Console.WriteLine($"Expensive product found:");

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

                Console.WriteLine($"Price: ₹{expensive.Price}");

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

            }

            else

            {

                Console.WriteLine("No expensive products found!");

            }

            Console.WriteLine();

        }

        // 4. Find Products by Category

        static async Task FindProductsByCategoryAsync(AppDbContext context)

        {

            Console.WriteLine("=== 4. FIND PRODUCTS BY CATEGORY ===");

            // Find all electronics products

            var electronicsProducts = await context.Products

                .Include(p => p.Category)

                .Where(p => p.Category.Name == "Electronics")

                .ToListAsync();

            Console.WriteLine($"Electronics products ({electronicsProducts.Count}):");

            foreach (var product in electronicsProducts)

            {

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

            }

            Console.WriteLine();

        }

        // 5. Display Categories with Product Count

        static async Task DisplayCategoryWithProductsAsync(AppDbContext context)

        {

            Console.WriteLine("=== 5. CATEGORIES WITH PRODUCT COUNT ===");

            var categories = await context.Categories

                .Include(c => c.Products)

                .ToListAsync();

            Console.WriteLine($"{"Category",-15} {"Products Count",-15}");

            Console.WriteLine(new string('-', 35));

            foreach (var category in categories)

            {

                Console.WriteLine($"{category.Name,-15} {category.Products.Count,-15}");

            }

            Console.WriteLine();

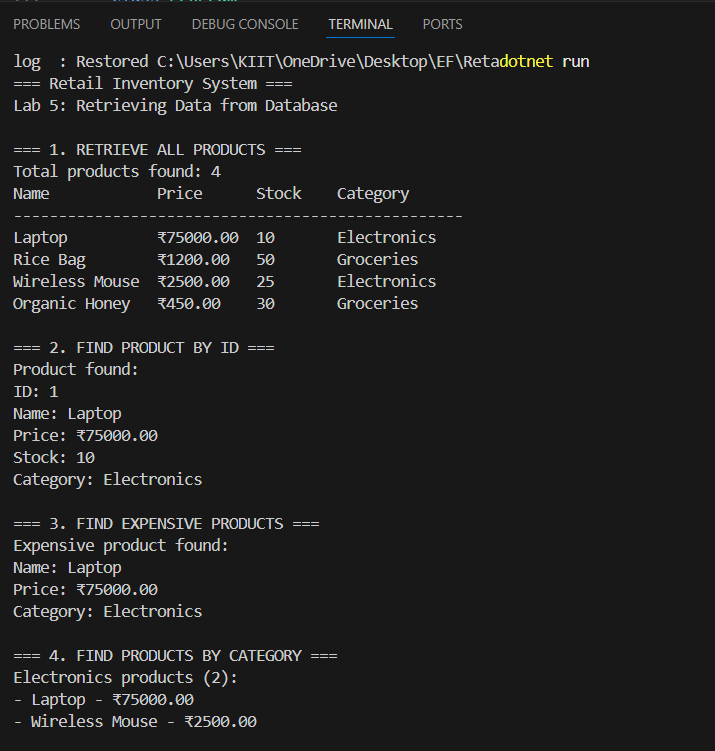
        }

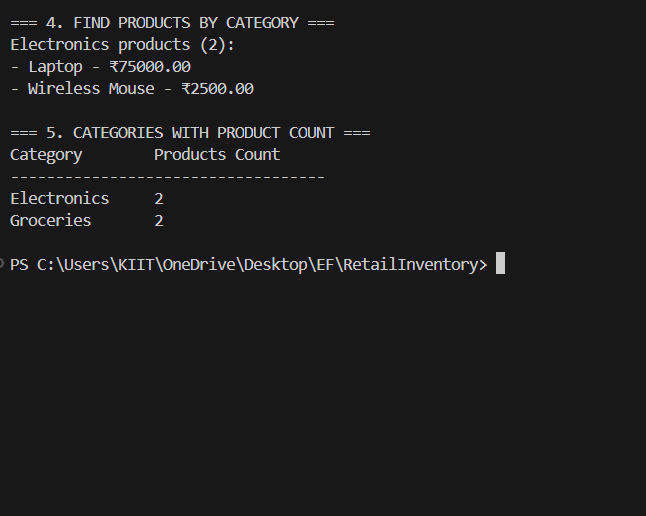
    }

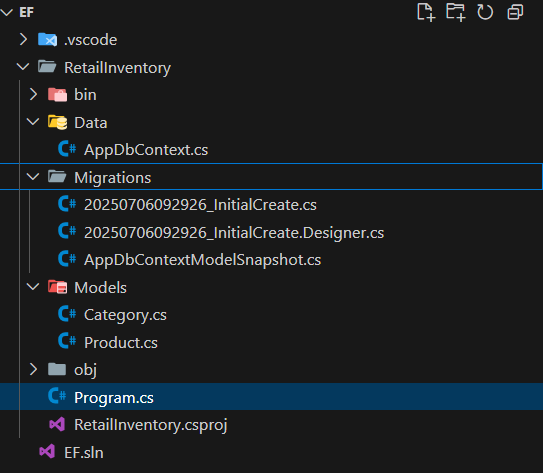
}

1. **Run the application:**

**dotnet run**

**Output:**



**Complete Project Structure:**

**Summary of Learning Objectives Achieved**

**Lab 1: Understanding ORM**

* Learned what ORM is and its benefits.
* Understood the differences between EF Core and EF Framework.
* Explored new features in EF Core 8.0.

**Lab 2: Database Context Setup**

* Created entity models with proper relationships.
* Configured the database context and connection string.
* Set up navigation properties.

**Lab 3: Database Migrations**

* Used EF Core CLI to create and apply migrations.
* Verified database schema creation.

**Lab 4: Data Insertion**

* Used AddAsync and AddRangeAsync methods.
* Implemented SaveChangesAsync for persisting data.
* Established relationships between entities.

**Lab 5: Data Retrieval**

* Used ToListAsync() to get all records.
* Used FindAsync() and FirstOrDefaultAsync() for specific queries.
* Used Include() to load related data.