6363257_Raunak

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

1. What is ORM ?

=> ORM (Object-Relational Mapping) is a technique that maps C# classes to database tables.

* Benefits: Productivity, maintainability, and abstraction from SQL. Benefits: -

- **Productivity**: No need to write repetitive SQL queries.

- **Maintainability:** Changes to the model are reflected in the database via migrations.

- **Abstraction**: Developers can focus on business logic instead of database syntax.

* Explain how ORM maps C# classes to database tables.

=> ORM (Object-Relational Mapping) is a programming technique that helps map C# classes to relational database tables. It allows developers to interact with the database using objects in C# instead of writing raw SQL queries. Entity Framework Core (EF Core) is an ORM for .NET that automates this process.

Example of Mapping :-

```
public class Product
{
public int Id { get; set; } public string N
ame { get; set; }
public decimal Price { get; set; }
}
```

When using EF Core, this class will automatically be mapped to a SQL table like this:-

SQL Table: Products

Id (int)	Name (nvarchar)	Price (decimal)
1	Laptop	75000.00

2. EF Core vs EF Framework

Feature	EF Core	EF Framework (EF6)
Platform Support	Cross-platform	Windows only
Performance	High	Moderate
Async Query Support	Yes	Limited
LINQ Support	Advanced	Basic
Compiled Queries	Yes	No
JSON Column Support	Yes (EF Core 8)	No
Best Use Case	New, modern .NET applications	Legacy .NET Framework apps

3. EF Core 8.0 Features

I. JSON Column Mapping.
EF Core 8.0 provides native support for mapping JSON columns in SQL Server. This feature enables developers to store structured data in JSON format within a single column, while still being able to query and manipulate the data using EF Core LINQ queries.
Use Case: Ideal for storing dynamic or hierarchical data such as product specifications, user preferences, or metadata without requiring schema changes.

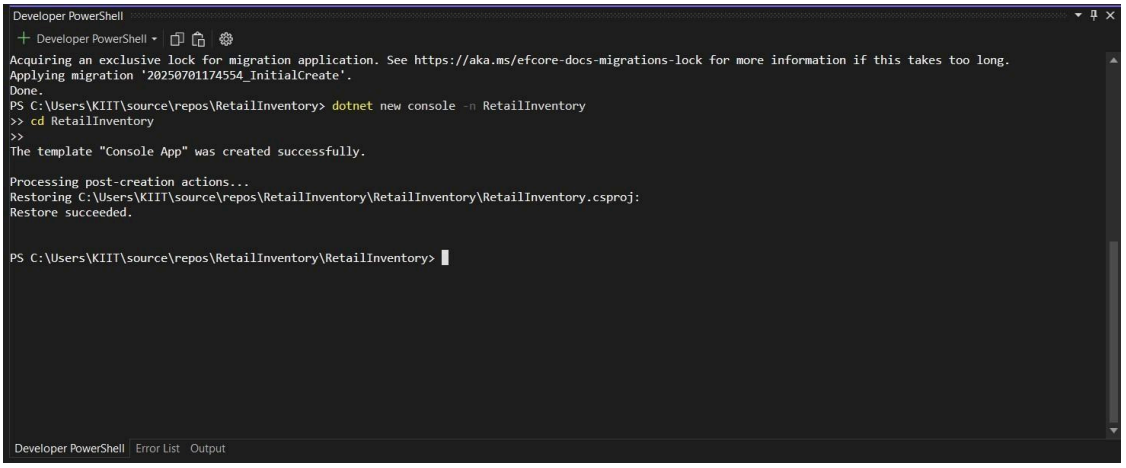
II. Compiled Models for Enhanced Performance.
Compiled models reduce the runtime cost of model building by allowing the application to precompile the EF Core model during build time. This significantly improves application startup time and reduces CPU usage in high-load scenarios.
Benefit: Faster startup performance and better scalability for large-scale applications.

III. Interceptors for Advanced Query and Command Handling
EF Core 8.0 introduces improved support for interceptors, which allow developers to intercept and customize EF Core operations such as query execution, data saving, or database connection management.
Use Case: Logging database operations, enforcing security rules, or injecting tenant-based filters in multi-tenant applications.

IV. Enhanced Bulk Operations.
With EF Core 8.0, bulk operations such as insert, update, and delete are optimized to handle large volumes of data more efficiently. This reduces the overhead of looping through records and executing individual operations.
Benefit: High-performance data manipulation, particularly useful for batch processing, data synchronization, and ETL scenarios.

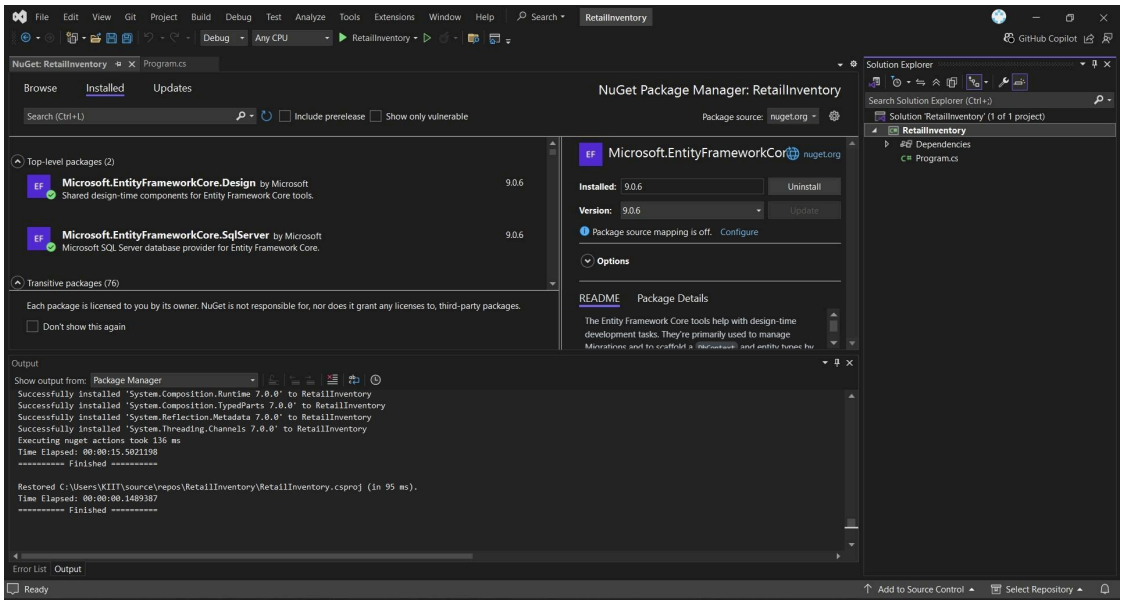
4 . Create a .NET Console App:-

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



5. Install EF Core Packages.

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



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:

I. Create

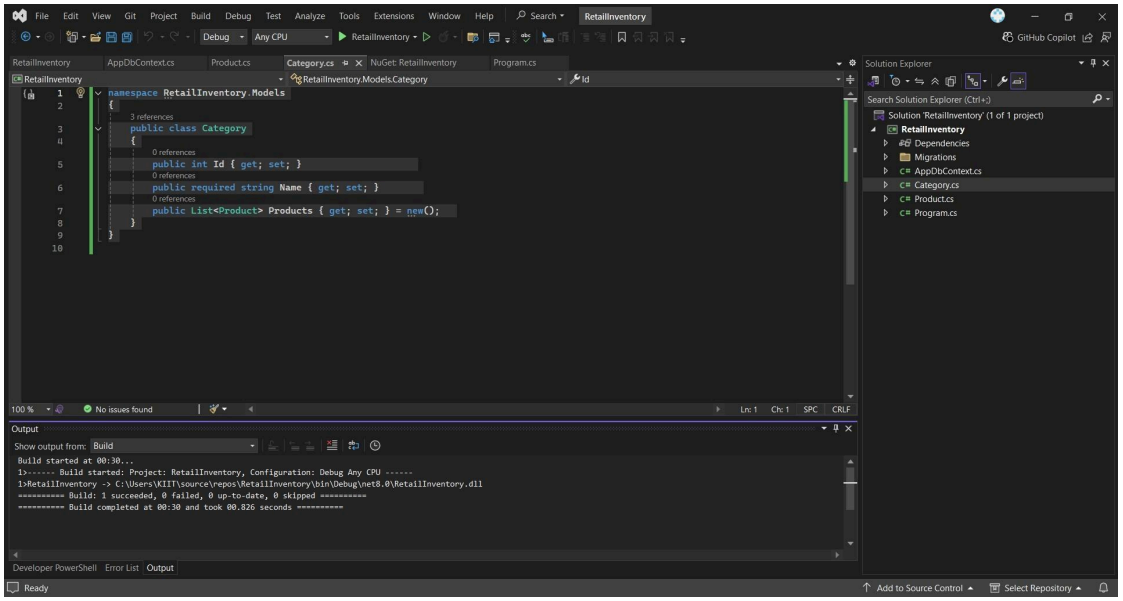
Models

Category.cs

Code :-

```
namespace RetailInventory.Models
{
	public class Category
	{
		public int Id { get; set; }
		public required string Name { get; set; }
		public List<Product> Products { get; set; } = new();
	}
}
```

OUTPUT:-

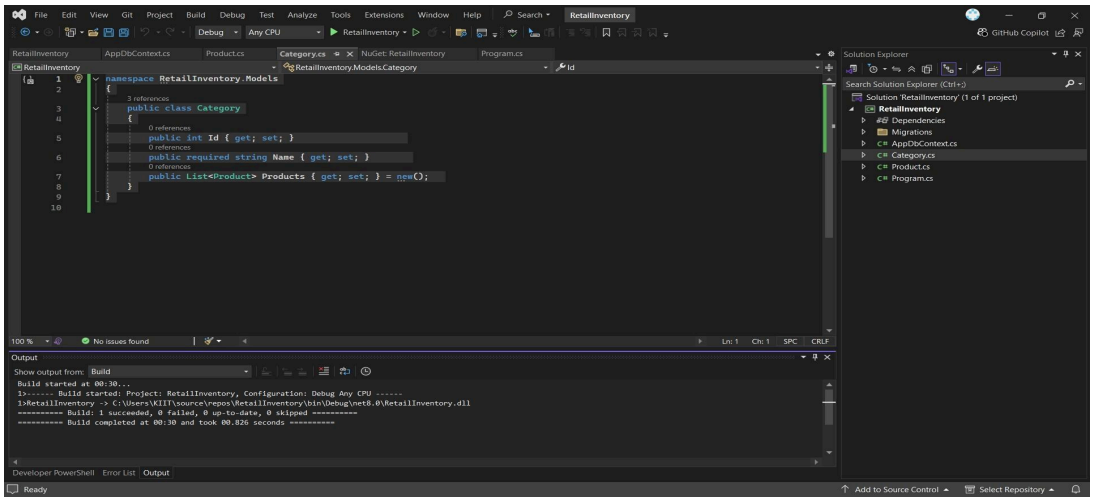


Product.cs

CODE:-

```
namespace RetailInventory.Models
{
	public class Product
	{
		public int Id { get; set; }
		public required string Name { get; set; } public decimal
		Price { get; set; }
		public int CategoryId { get; set; }
		public required Category Category { get; set; }
	}
}
```

OUTPUT:-

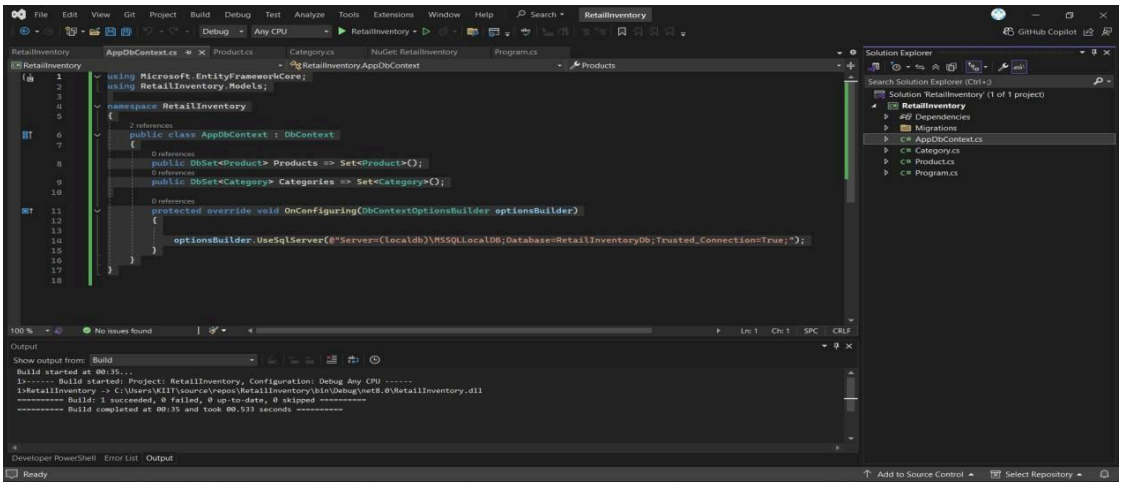


II. Create AppDbContext:

CODE :-

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

namespace RetailInventory
{
	public class AppDbContext : DbContext
	{
		public DbSet<Product> Products => Set<Product>(); public
		DbSet<Category> Categories => Set<Category>();

		protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
		{

optionsBuilder.UseSqlServer(@"Server=(localdb)\MSSQLlocalDB;Database=Retai
llInventoryDb;Trusted_Connection=True;");
		}
	}
}
```

OUTPUT:-



III. add Connection String in appsettings.json (optional for ASP.NET Core).

To make the database connection configurable, we added a JSON file named appsettings.json to the project. This file stores the SQL Server connection string in a structured format.

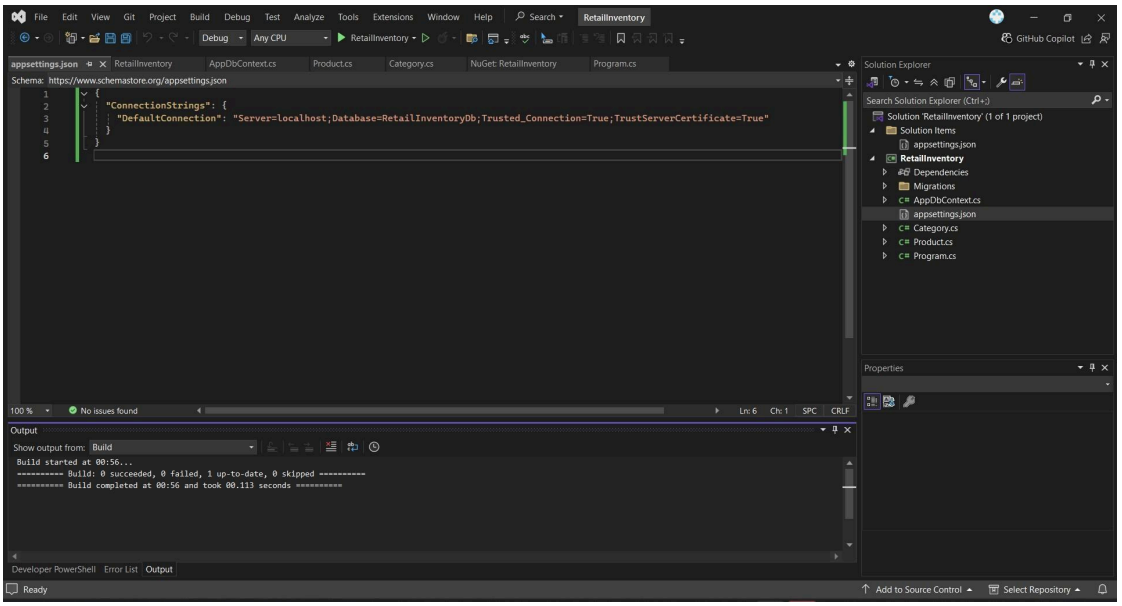
JSON Content:

appsettings.json

CODE :-

```
{
   "ConnectionStrings": {
      "DefaultConnection":
"Server=localhost;Database=RetailInventoryDb;Trusted_Connection=True;Trust
ServerCertificate=True"
   }
}
```

OUTPUT:-



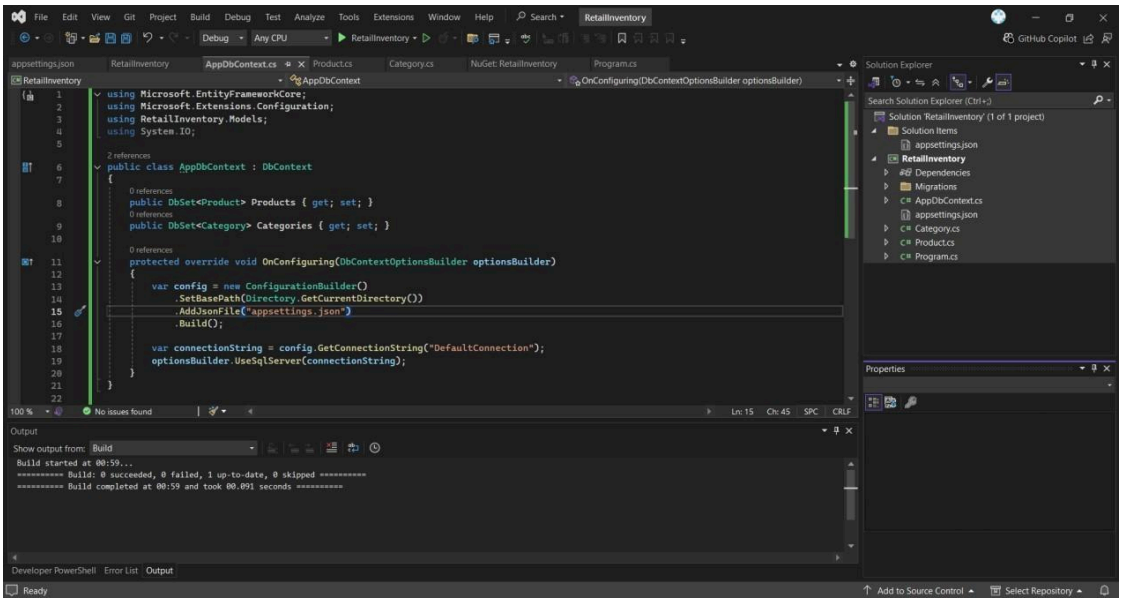
Configuration in AppDbContext.cs:

In the OnConfiguring() method of AppDbContext.cs, the following code reads the connection string from appsettings.json:

CODE:-

```
using Microsoft.EntityFrameworkCore; using
Microsoft.Extensions.Configuration; using
RetailInventory.Models;
using System.IO;

public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; } public
    DbSet<Category> Categories { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json")
            .Build();

        var connectionString = config.GetConnectionString("DefaultConnection");
        optionsBuilder.UseSqlServer(connectionString);
    }
}
```

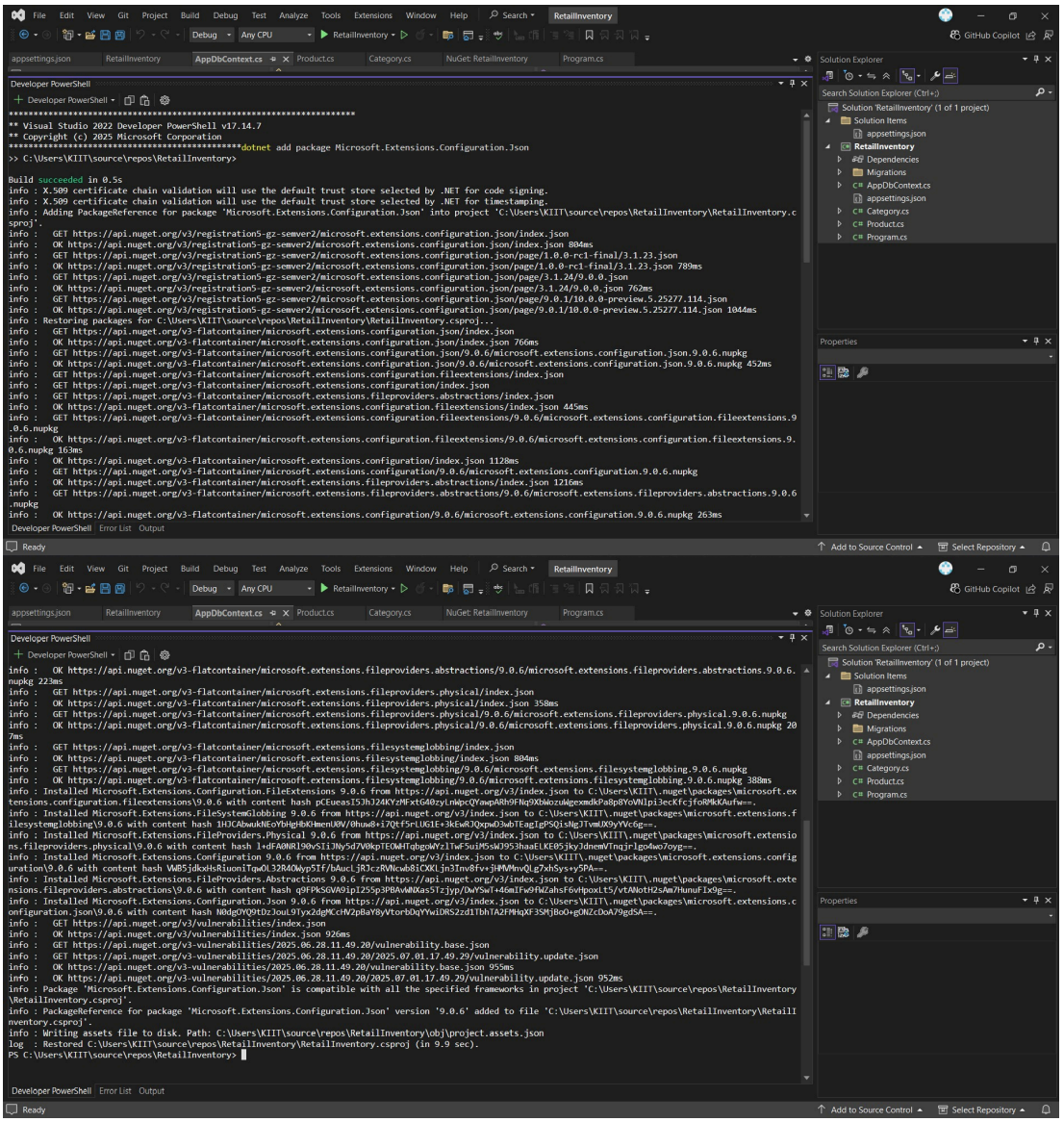
OUTPUT:-



Packages Installed:-

dotnet add package

Microsoft.Extensions.Configuration.Json OUTPUT:-



Lab 3: Using EF Core CLI to Create and Apply Migration

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:-

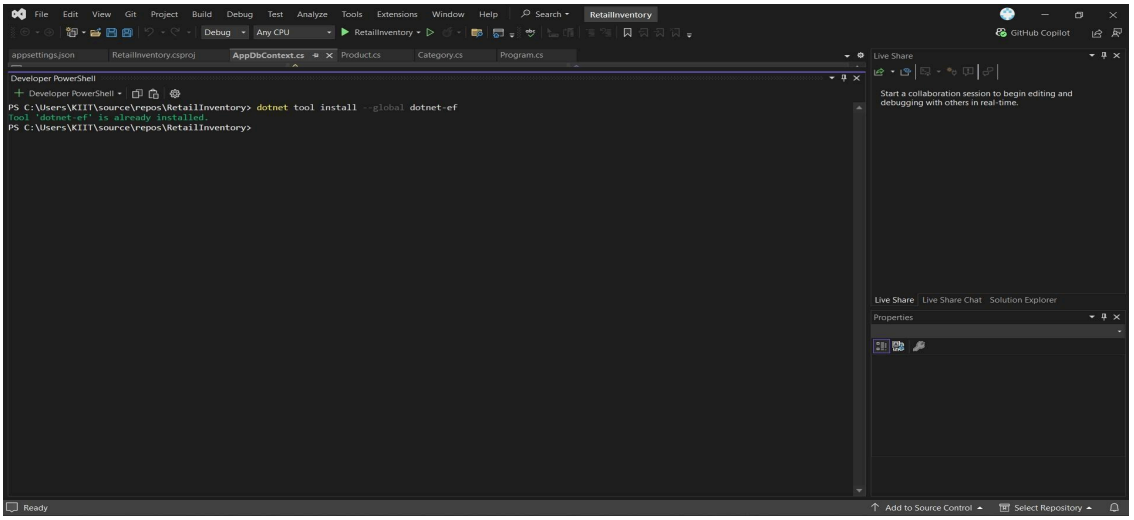
Learn how to use EF Core CLI to manage database schema changes.

Steps:-

I. Install EF Core CLI (if not already):-

dotnet tool install --global dotnet-e

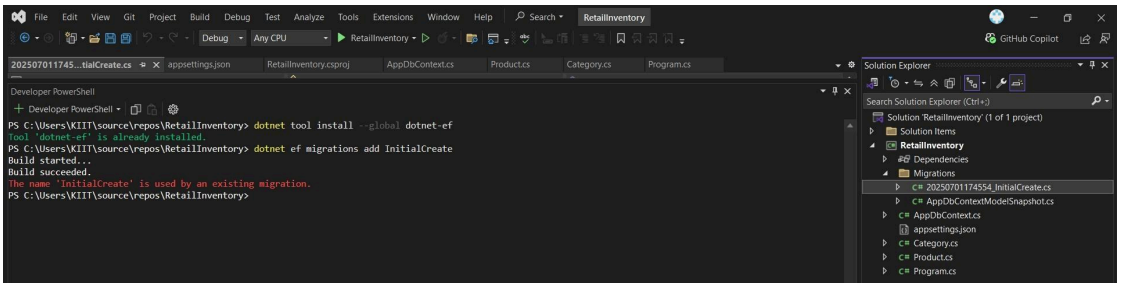
OUTPUT:-



II. Create Initial

Migration:- dotnet ef migrations

add InitialCreate OUTPUT:-

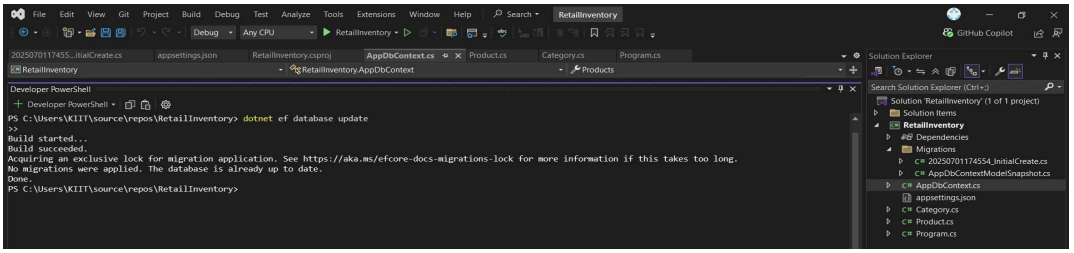


This generates a Migrations folder with code that represents the schema based on the models I previously created in Category.cs, Product.cs, and AppDbContext.cs.

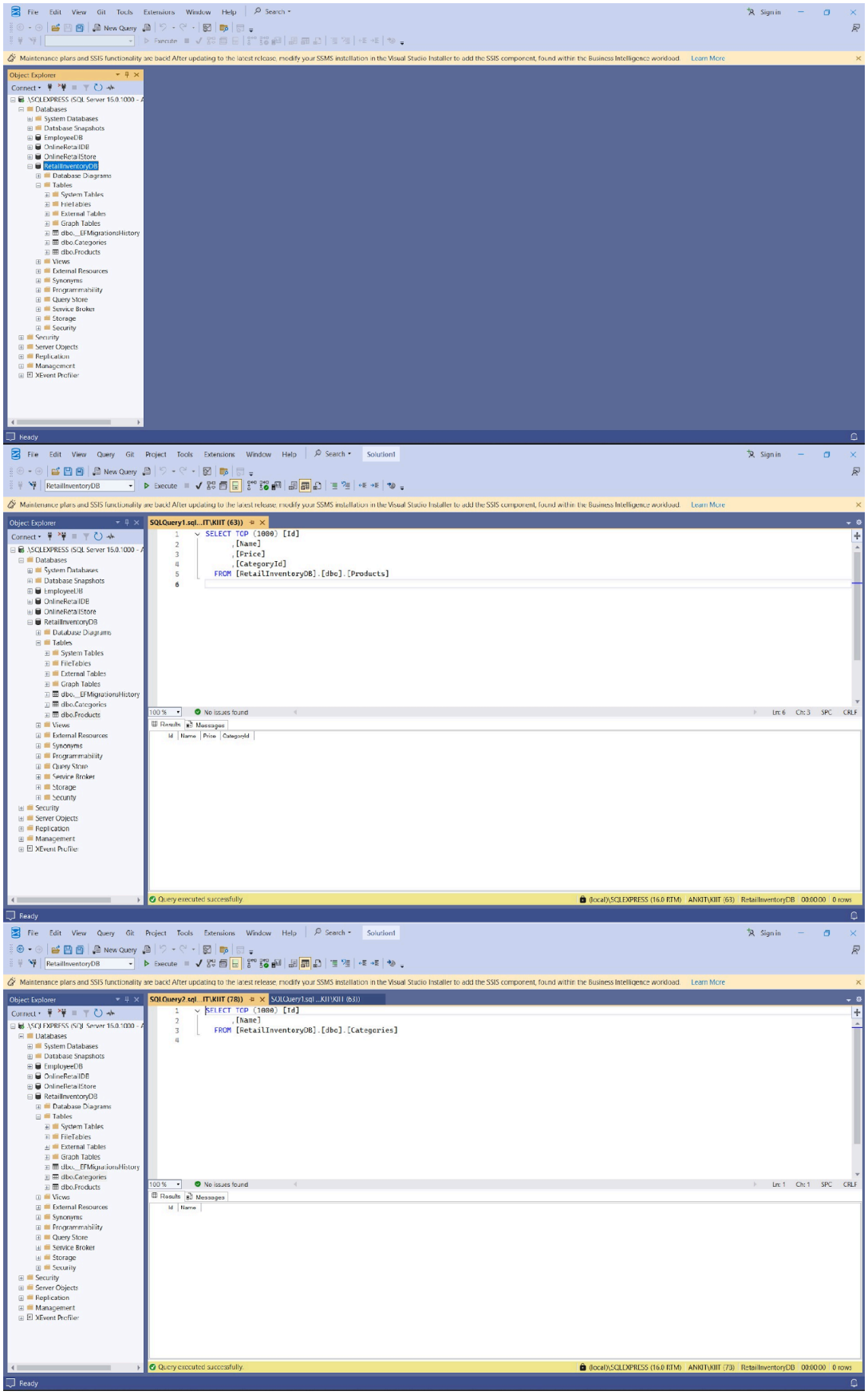
III. Apply Migration to Create Database:-

dotnet ef database update

OUTPUT:-



IV. Verify in SQL Server:-



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.

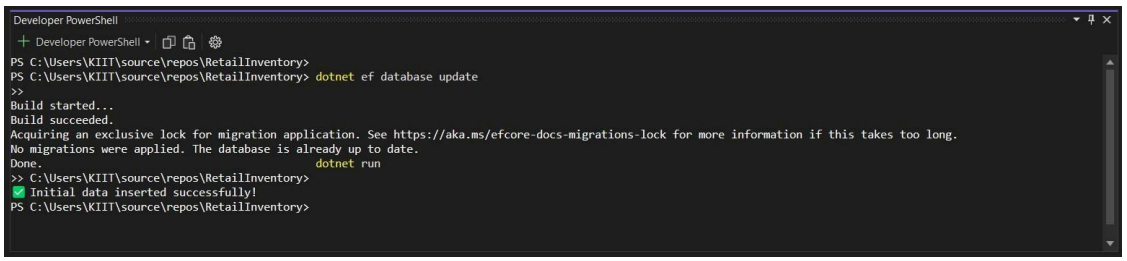
Steps:-

I. Insert Data in Program.cs:-

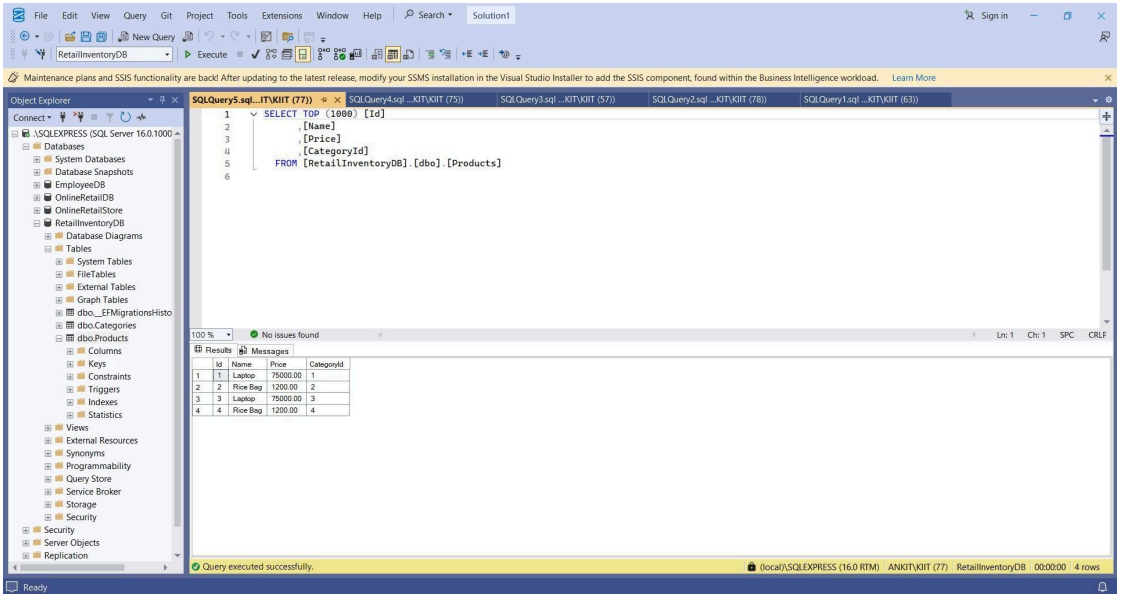
CODE:-

```
using RetailInventory;
using RetailInventory.Models; using
System;
using System.Threading.Tasks;

class Program
{
      static async Task Main(string[] args)
      {
            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 = electronics };
            var product2 = new Product { Name = "Rice Bag", Price = 1200, Category =
groceries };

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

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

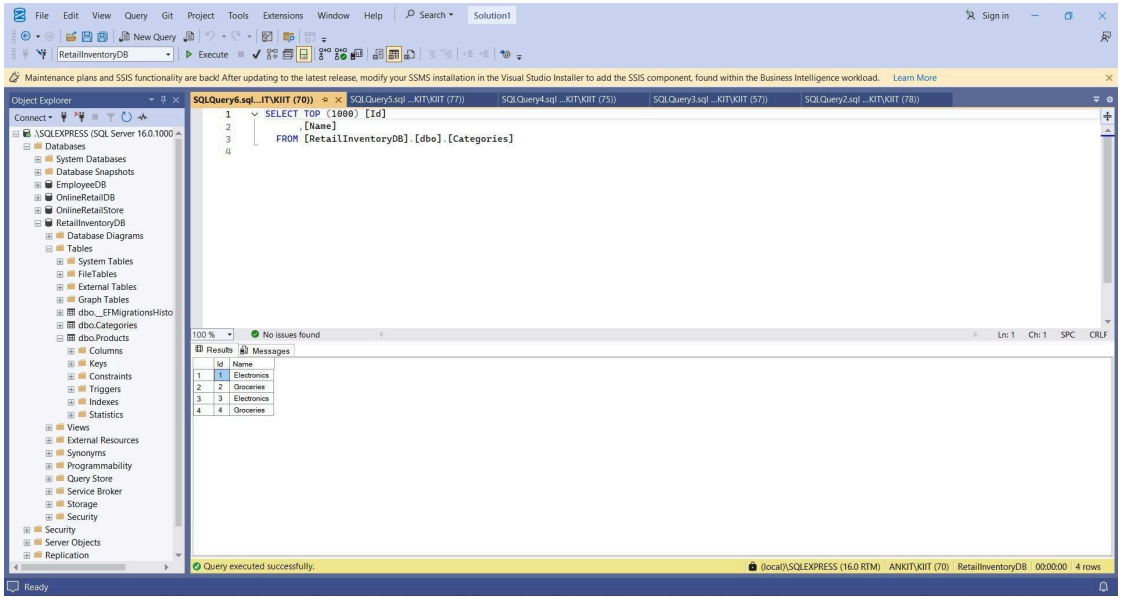
OUTPUT:-



The Below Screenshot showing successful insertion of products into RetailInventoryDB's Products table.



The Below Screenshot showing successful insertion of products into RetailInventoryDB's Categories table.



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.

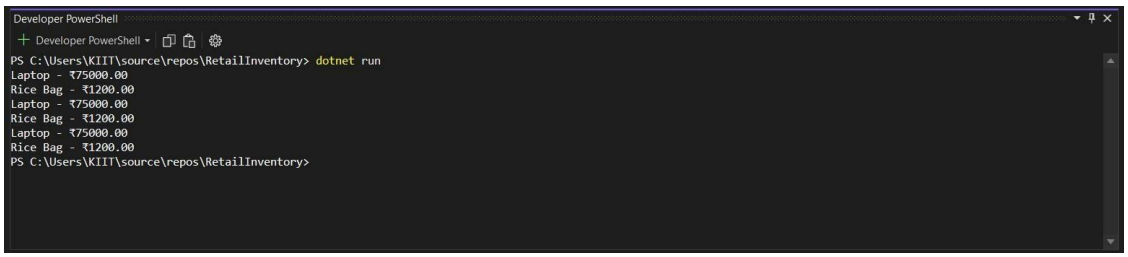
Steps:-

I. Retrieve All

Products:- CODE:-

```
using Microsoft.EntityFrameworkCore; using
RetailInventory;
using RetailInventory.Models; using
System;
using System.Threading.Tasks;

class Program
{
	static async Task Main(string[] args)
	{

		using var context = new AppDbContext();
		var products = await context.Products.ToListAsync(); foreach (var p in
		products)
			Console.WriteLine($"{p.Name} - ₹{p.Price}");
	}
}
```

OUTPUT:-



II. Find by

ID:- CODE:-

```
using Microsoft.EntityFrameworkCore; using
RetailInventory;
using RetailInventory.Models; using
System;
using System.Threading.Tasks;

class Program
{
	static async Task Main(string[] args)
	{
		using var context = new AppDbContext();
		var product = await context.Products.FindAsync(1); Console.WriteLine($"Found:
		{product?.Name}");
	}
}
```

OUTPUT:-

III. FirstOrDefault with

Condition:- CODE:-

```
using Microsoft.EntityFrameworkCore; using
RetailInventory;
using RetailInventory.Models; using
System;
using System.Threading.Tasks;

class Program
{
      static async Task Main(string[] args)
      {
            using var context = new AppDbContext();
            var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 50000);
            Console.WriteLine($"Expensive: {expensive?.Name}");
      }
}
```

OUTPUT:-