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.

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

=> ORM (Object-Relational Mapping) is a programming technique that allows developers to interact with a relational database using object-oriented code, rather than writing raw SQL queries.

ORM maps C# classes to database table in following way:-

1. C# classes representing data entities are defined, with properties corresponding to database columns.
2. ORM frameworks use configuration files or attributes to specify how classes map to database tables. This can include mapping class names to table names, property names to column names, and data types
3. ORMs can use a "code-first" approach, where the class definitions are the source of truth and the database schema is generated based on them. Alternatively, a "database-first" approach starts with an existing database schema and generates corresponding classes.
4. ORM frameworks handle relationships between classes, such as one-to-many or many-to-many, by mapping them to foreign key constraints and join tables in the database.
5. ORM frameworks provide APIs for performing CRUD (Create, Read, Update, Delete) operations on objects, which are then translated into corresponding SQL queries by the ORM.
6. Entity Framework Core, a popular .NET ORM, allows you to define your classes and then use DbContext to manage the database context. Code-first migrations can then be used to create the database tables based on your class definitions.

Benefits:-

1. Eliminates the need to write repetitive SQL queries.
2. ORMs can abstract database-specific syntax, allowing for easier switching between different database systems.
3. Developers work with objects, making data manipulation more intuitive and efficient.

**Steps: 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.

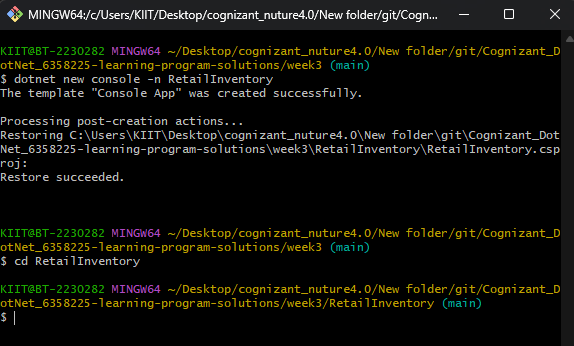
**Steps: 3** EF Core 8.0 Features:

• JSON column mapping.

• Improved performance with compiled models.

• Interceptors and better bulk operations.

**Steps: 4** Create a .NET Console App:

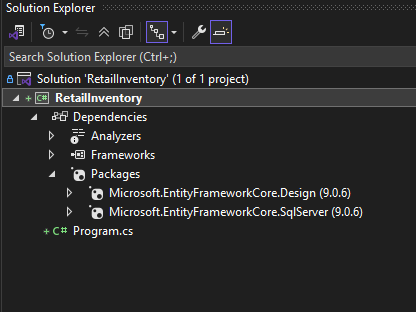


**Steps: 5**  Install EF Core Packages:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Design

We can see the dependencies are installed:-



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.

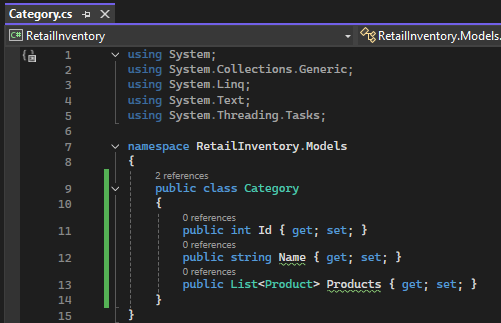
Objective:

Configure DbContext and connect to SQL Server.

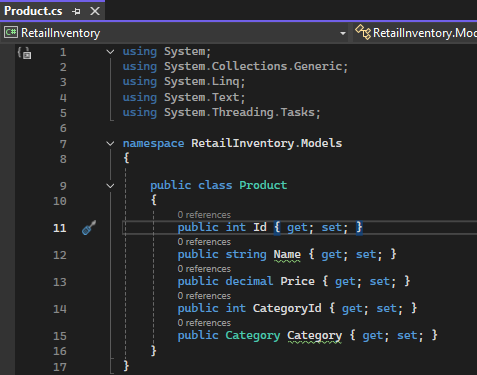
Steps:

1. Create Models:

Screenshot of Category Entity class:-



Screenshot of Product Entity class:-

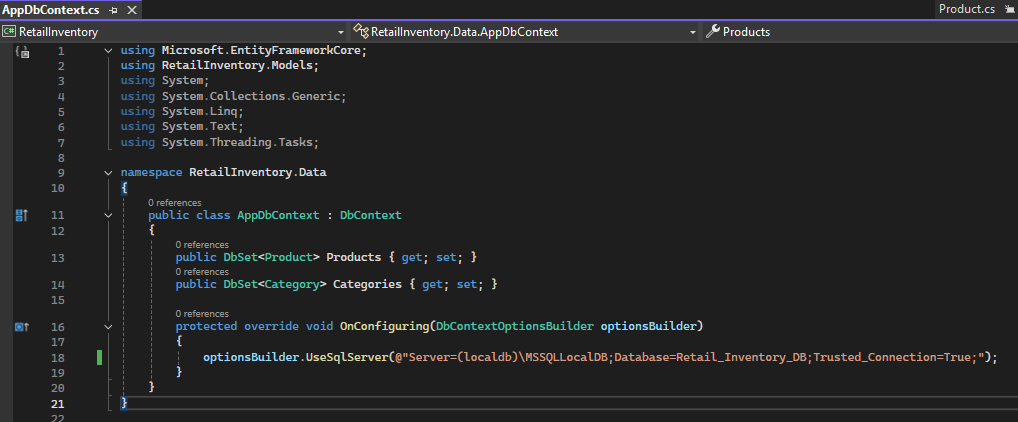


1. Create AppDbContext:

Connection string: @"Server=(localdb)\MSSQLLocalDB;Database=Retail\_Inventory\_DB;Trusted\_Connection=True;"

Here Database: Retail\_Inventory\_DB

Screenshot of AppDbcontext class with connection string



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

Note: I have already installed ‘dotnet-ef’ tool

1. Create Initial Migration:

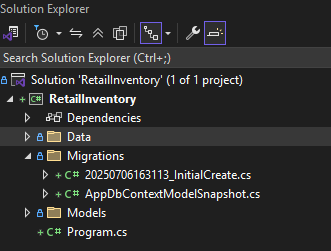
dotnet ef migrations add InitialCreate

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

3. Apply Migration to Create Database:

dotnet ef database update

Screenshot of Migration folder with InitialCreate and update:-

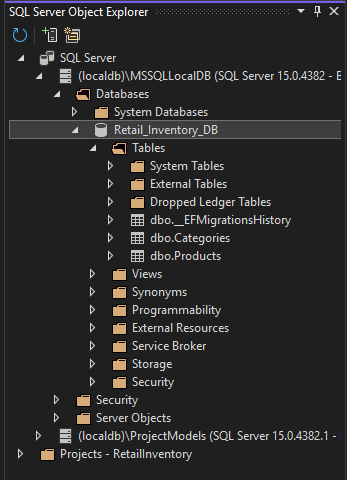


1. Verify in SQL Server:

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

that tables Products and Categories are created.

Screenshot verifying the database named “Retail\_Inventory\_DB” with tables Product and Category is created and can be seen under SQL Server object explorer



Lab 4: Inserting Initial Data into the Database

Scenario:

The store manager wants to add initial product categories and products to the system.

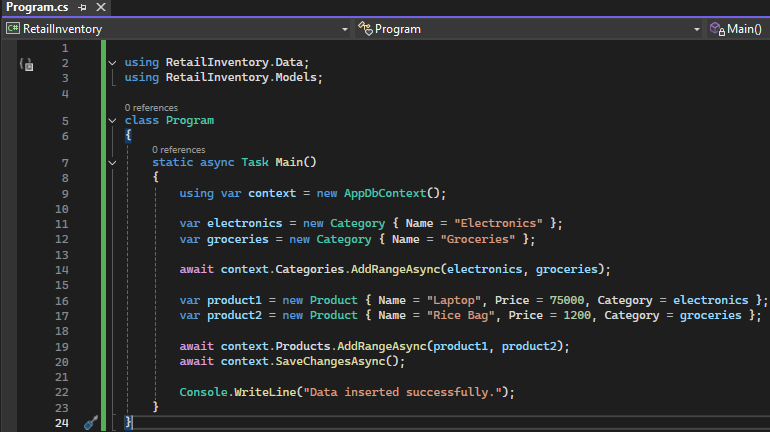
Objective:

Use EF Core to insert records using AddAsync and SaveChangesAsync.

Steps:

1. Insert Data in Program.cs:

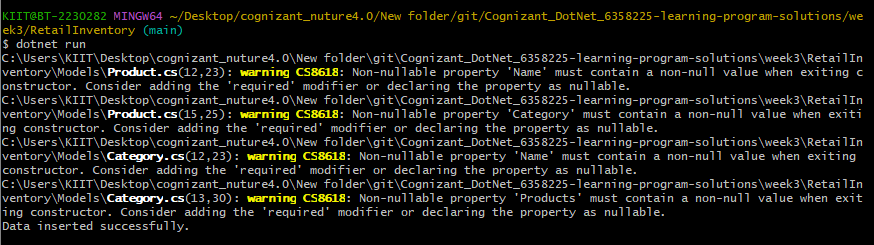
Screenshot of code:-



2. Run the App:

dotnet run

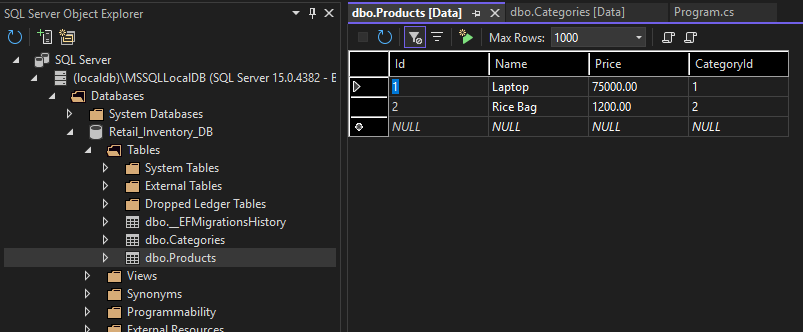
Screenshot of output:-



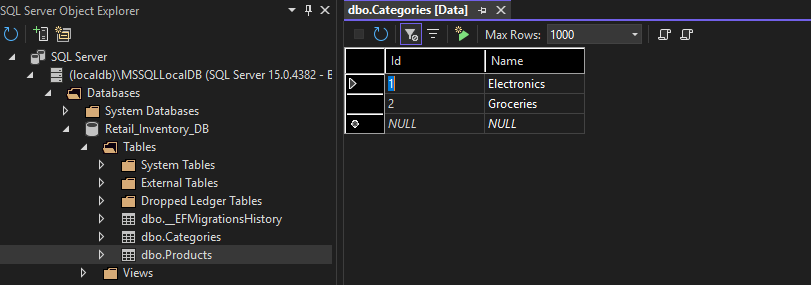
We can see it prints ‘Data Inserted successfully’.

1. Verifying in Sql Server:-

Screenshot displaying the Product table:



Screenshot displaying the Category 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:

1. Retrieve All Products:

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

foreach (var p in products)

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

1. Find by ID:

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

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

1. FirstOrDefault with Condition:

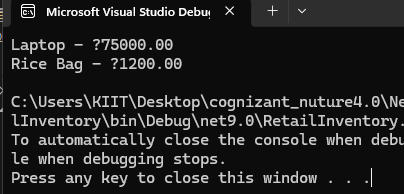
var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 50000);

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

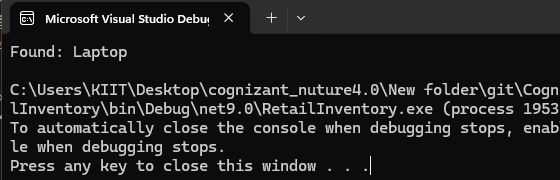
We first create the instance of AppContextDb named ‘context’:-

using var context = new AppDbContext();

Output Screenshot of retrieve all products:-



Output Screenshot of find by Id:-



Output Screenshot of First or Default with condition:-

