**Week 3 - Entity Framework Core 8.0 - Hands-on**

DEBANJAN KAURI(Superset ID-6362388)

**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.

To achieve this:

1. What is ORM? (Explain with Benefits)

2. EF Core vs EF Framework

3. Create a .NET Console App

4. Install EF Core Packages

**Explanation:**

Object-Relational Mapping (ORM) is a technique that connects object-oriented programming with relational databases. It maps C# classes to database tables and their properties to corresponding table columns. This allows developers to interact with databases using C# code instead of writing raw SQL, making data access more intuitive and maintainable.

Benefits:

1. Boosts productivity (less manual SQL).

2. Improves maintainability (models reflect DB schema).

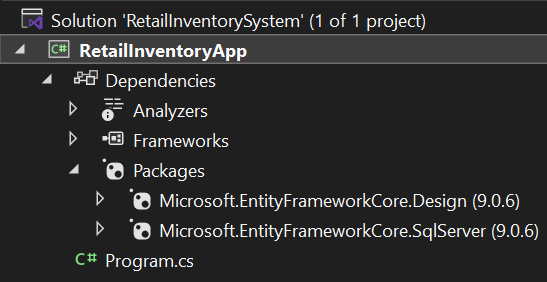
3. Provides abstraction from SQL queries.

4. Enables type-safe and LINQ-based querying.

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.

### **Console App:**

j

**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.

To achieve this:

1. Create Models (with Category & Product class)

2. Create AppDbContext.

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

### **Code: (In Models)**

*Category.cs*

using System.Collections.Generic;

namespace RetailInventoryApp.Models

{

public class Category

{

public int Id { get; set; }

public required string Name { get; set; }

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

}

}

*Product.cs*

using System;

using System.Collections.Generic;

namespace RetailInventoryApp.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; }

}

}

### **AppDbContext:** *RetailDbContext.cs*

using Microsoft.EntityFrameworkCore;

using RetailInventoryApp.Models;

namespace RetailInventoryApp

{

public class RetailDbContext : DbContext

{

public DbSet<Product> Products => Set<Product>();

public DbSet<Category> Categories => Set<Category>();

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

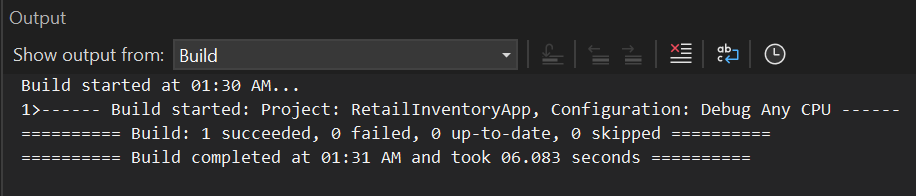
optionsBuilder.UseSqlServer("Server=BT-22051676;Database=RetailDB;Trusted\_Connection=True;Encrypt=False;");

}

}

}

### **Output:**



**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.

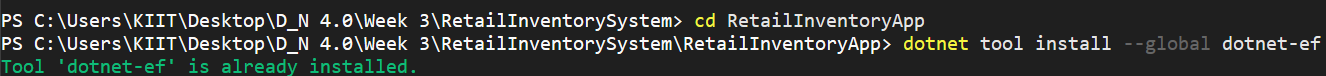
To achieve this:

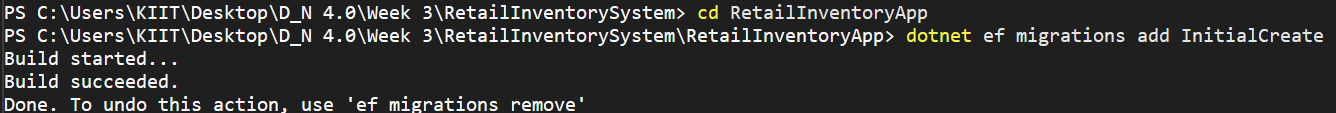
1. Install EF Core CLI (if not already).

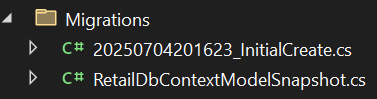
2. Create Initial Migration.

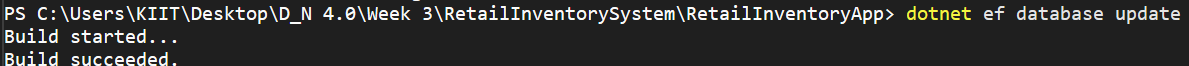
3. Apply Migration to Create Database.

4. Verify in SQL Server.

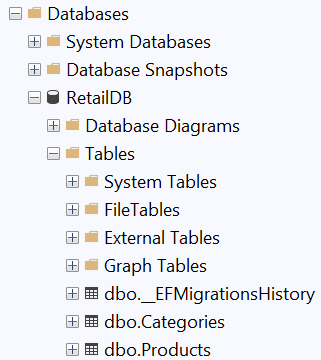
**EF Core CLI (***installation***):**

**Initial Migration:**



**Apply Migration:**

**Verify 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.

To achieve this:

1. Insert Data in Program.cs.

2. Run the App.

3. Verify in SQL Server.

**Code:** *Program.cs*

using RetailInventoryApp.Models;

using System;

namespace RetailInventoryApp

{

internal class Program

{

static void Main(string[] args)

{

using var context = new RetailDbContext();

if (!context.Categories.Any())

{

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

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

context.Categories.AddRange(electronics, groceries);

context.SaveChanges();

var product1 = new Product

{

Name = "Laptop",

Price = 75000,

Category = electronics

};

var product2 = new Product

{

Name = "Rice Bag",

Price = 1200,

Category = groceries

};

context.Products.AddRange(product1, product2);

context.SaveChanges();

Console.WriteLine(" Sample data inserted into database.");

}

else

{

Console.WriteLine(" Database already contains data. Skipping insertion.");

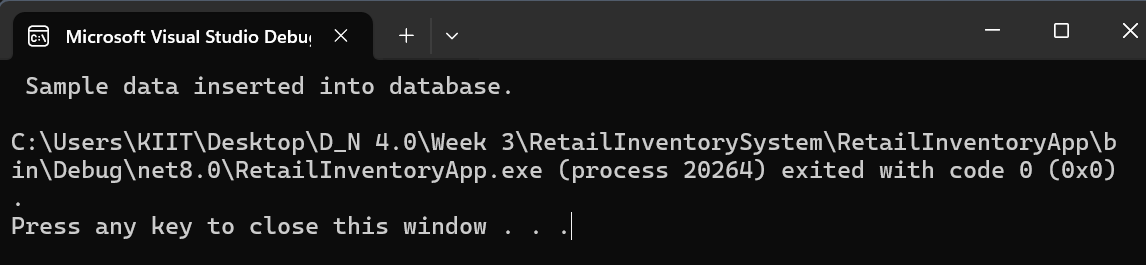
}

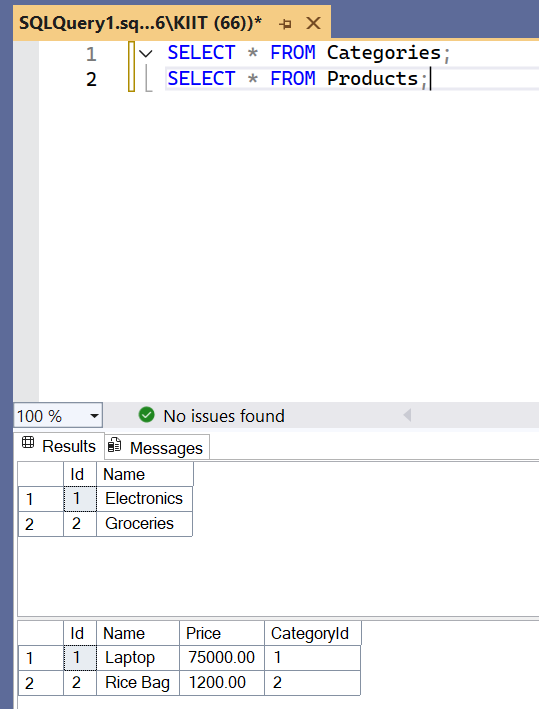
}

}

}

**Run the App:**



**Verify in SQL Server:**

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

Scenario:

The store wants to display product details on the dashboard.

To achieve this:

1. Retrieve All Products.

2. Find by ID.

3. FirstOrDefault with Condition.

**Code:**

using System;

using System.Linq;

using Microsoft.EntityFrameworkCore;

using RetailInventoryApp.Models;

namespace RetailInventoryApp

{

internal class Program

{

static void Main(string[] args)

{

using var context = new RetailDbContext();

if (!context.Products.Any())

{

Console.WriteLine(" No product data found. Please insert data using Lab 4.");

return;

}

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

Console.WriteLine(" Products with Categories:");

Console.WriteLine("-----------------------------");

foreach (var product in products)

{

Console.WriteLine($" Product: {product.Name,-15} | Rs:{product.Price,-8} | Category: {product.Category?.Name}");

}

var productById = context.Products.Find(1);

Console.WriteLine("\n Product with ID = 1:");

Console.WriteLine(productById != null ? $"> {productById.Name}" : "Not found.");

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

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

Console.WriteLine(expensive != null ? $"> {expensive.Name}" : "No expensive product found.");

Console.WriteLine("\nQuery completed.");

}

}

}

### **Output:**

## *Additional Hands-On*

**Lab 6: Updating and Deleting Records**

Scenario:

The store updates product prices and removes discontinued items.

To achieve this:

1. Update a Product.

2. Delete a Product.

**Code:**

using System;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using RetailInventoryApp.Models;

namespace RetailInventoryApp

{

internal class Program

{

static async Task Main(string[] args)

{

using var context = new RetailDbContext();

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

if (productToUpdate != null)

{

Console.WriteLine($"Updating price of {productToUpdate.Name} from Rs.{productToUpdate.Price} to Rs.70000...");

productToUpdate.Price = 70000;

await context.SaveChangesAsync();

Console.WriteLine("Product price updated successfully.");

}

else

{

Console.WriteLine("Product 'Laptop' not found.");

}

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

if (productToDelete != null)

{

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

context.Products.Remove(productToDelete);

await context.SaveChangesAsync();

Console.WriteLine("Product deleted successfully.");

}

else

{

Console.WriteLine("Product 'Rice Bag' not found.");

}

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

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

foreach (var product in remainingProducts)

{

Console.WriteLine($"- {product.Name}, ₹{product.Price}, Category: {product.Category?.Name}");

}

}

}

}

### **Output:**

**Lab 7: Writing Queries with LINQ**

Scenario:

The store wants to filter and sort products for reporting.

To achieve this:

1. Filter and Sort.

2. Project into DTO.

**Code:**

using System;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using RetailInventoryApp.Models;

namespace RetailInventoryApp

{

internal class Program

{

static async Task Main(string[] args)

{

using var context = new RetailDbContext();

var filtered = await context.Products

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

.OrderByDescending(p => p.Price)

.ToListAsync();

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

foreach (var p in filtered)

{

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

}

var productDTOs = await context.Products

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

.ToListAsync();

Console.WriteLine("\nProjected Products (DTO):");

foreach (var dto in productDTOs)

{

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

}

}

}

}

### **Output:**