Entity Framework Core Handson

Superset id: 6363535

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

### **What is ORM?**

**ORM** stands for **Object-Relational Mapping**. It is a technique that allows developers to interact with a database using the programming language's objects, instead of writing raw SQL queries. ORM tools automatically map database tables to programming language classes and rows to objects.

### **How ORM Maps C# Classes to Database Tables**

In C#, ORM is typically handled using frameworks like **Entity Framework (EF)**. Here's how the mapping works:

| **C# Concept** | **Database Concept** |
| --- | --- |
| Class | Table |
| Property | Column |
| Object Instance | Row (Record) |
| Navigation Property | Foreign Key / Relationship |

### **Benefits of ORM**

1. **Productivity**: Developers can focus on business logic rather than SQL. CRUD operations are handled via method calls.
2. **Maintainability**: Clean, centralized model classes improve readability and reduce duplication.
3. **Abstraction from SQL**: Avoids boilerplate SQL and handles complex joins or transactions through LINQ and object navigation.
4. **Database-agnostic code**: Easily switch databases with minimal changes.
5. **Security**: Helps prevent SQL injection by using parameterized queries internally.

### **2. EF Core vs EF Framework (EF6)**

**Entity Framework Core (EF Core)** and **Entity Framework 6 (EF6)** are both Object-Relational Mapping (ORM) frameworks from Microsoft, but they differ in design, capabilities, and target platforms.

### **EF Core**

* **Cross-platform**: Works on Windows, Linux, and macOS.
* **Lightweight and modular**: Designed from scratch with performance in mind.
* **Modern features**:  
  + Full **LINQ** support.
  + **Async** operations using async/await.
  + **Compiled queries** for better performance.
* **Active development**: Continuously evolving with new features.
* **Better support for NoSQL and cloud-native architectures**.

### **Entity Framework 6 (EF6)**

* **Windows-only**: Built on .NET Framework.
* **Mature and stable**: Battle-tested in enterprise applications.
* **Fewer modern features**: Limited support for async and lacks some EF Core optimizations.
* **Rich designer support**: Includes EDMX (Entity Data Model) visual designer for easier modeling.
* **Better backward compatibility**: Preferred in legacy applications.

### 

### **3. EF Core 8.0 Features**

Entity Framework Core 8.0 introduces several powerful enhancements aimed at improving performance, flexibility, and real-world data handling. Here are some key features:

### **1. JSON Column Mapping**

* EF Core 8.0 allows mapping C# objects directly to **JSON columns** in relational databases (like SQL Server 2022 and PostgreSQL).
* You can query, filter, and update nested JSON data without needing a separate table structure.
* Ideal for handling semi-structured or hierarchical data within a relational model.

### **2. Compiled Models for Performance**

* EF Core 8.0 improves startup and query performance by supporting **precompiled models**.
* This avoids runtime model-building overhead, making it faster and more efficient—especially beneficial in large-scale or high-throughput apps.

### **3. Interceptors and Bulk Operations**

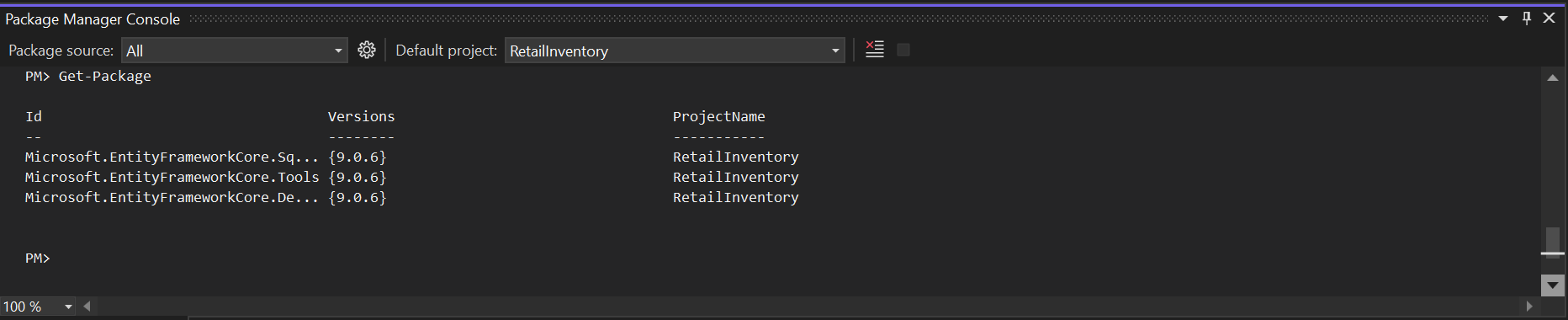
* **Interceptors** allow you to hook into query execution, command interception, and logging—giving better control over database behavior (e.g., auditing, logging, caching).
* **Improved bulk operations** support high-performance inserts, updates, and deletes in batches, reducing round trips to the database and improving throughput.

These features make EF Core 8.0 more robust and efficient for modern, high-performance applications.

**Install EF Core Packages:**

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Design

****

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

**Model.cs**public class Category{

public int Id { get; set; }

public string Name { get; set; }

public List<Product> 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; }

}

**AppDbContext.cs**

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

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);

}

}

**AppSettings.Json**

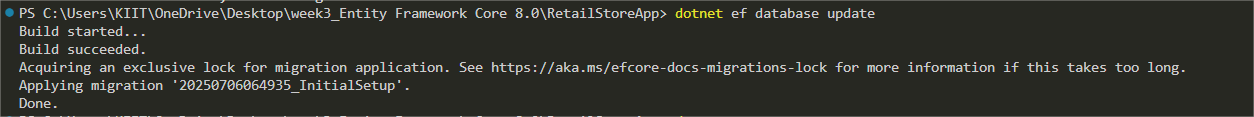
{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=RetailInventoryDB;Trusted\_Connection=True;"

}

}



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

PS C:\Users\KIIT\Desktop\week3\_EntityFrameworkCore\RetailInventory> dotnet tool install --global dotnet-ef

>>

Tool 'dotnet-ef' is already installed.

PS C:\Users\KIIT\Desktop\week3\_EntityFrameworkCore\RetailInventory> dotnet ef migrations add InitialCreate

>>

Build started...

Build succeeded.

Done. To undo this action, use 'ef migrations remove'

PS C:\Users\KIIT\Desktop\week3\_EntityFrameworkCore\RetailInventory> dotnet ef database update

>>

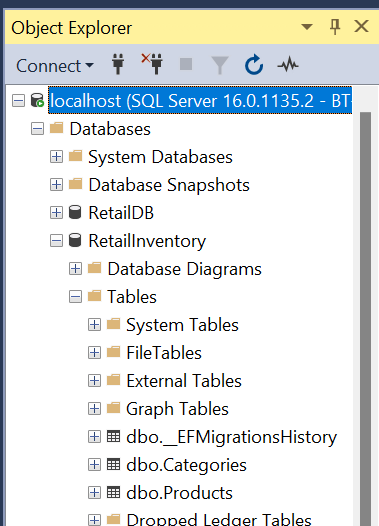
Build started...

Build succeeded.

Acquiring an exclusive lock for migration application. See https://aka.ms/efcore-docs-migrations-lock for more information if this takes too long.

Applying migration '20250705223539\_InitialCreate'.

Done.



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

**Program.cs**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);

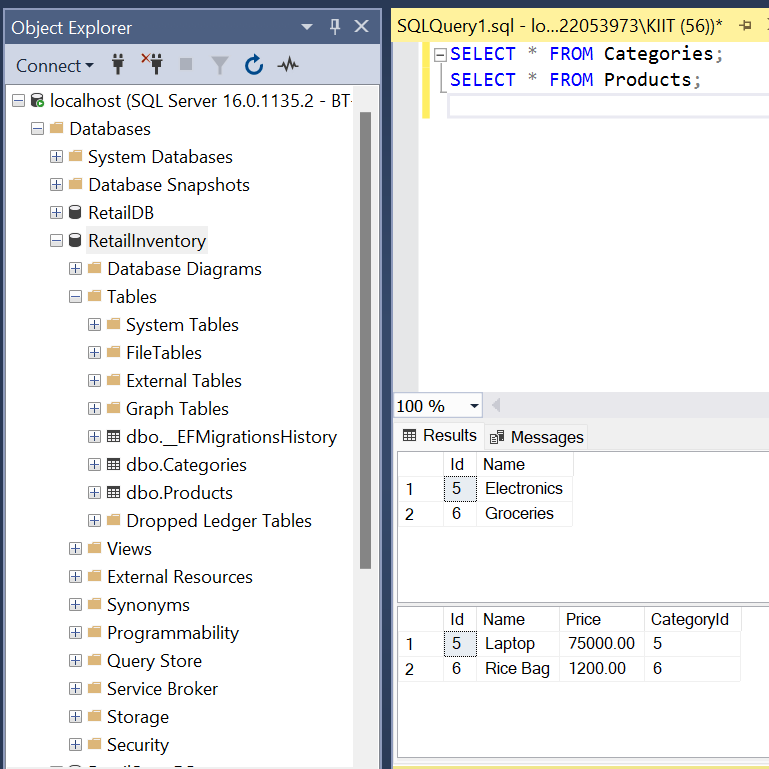
await context.SaveChangesAsync();

Console.WriteLine("Initial data inserted successfully....");

}

}

****

****

**Lab 5: Retrieving Data from the Database  
  
Program.cs**using System;

using System.Threading.Tasks;

using System.Linq;

using Microsoft.EntityFrameworkCore;

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("\nFind Product By ID:");

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

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

Console.WriteLine("\nFirst Product with Price > ₹50000:");

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

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

}

}

****