**Entity Framework Core 8.0 Hands-on**

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

**Step 1: What is ORM?**

Object-Relational Mapping (ORM) is a technique to map objects in your code (C# classes) to relational database tables.

#### **Benefits of ORM:**

* **Productivity**: You write C# code instead of SQL.
* **Maintainability**: Centralized model definitions.
* **Abstraction**: Reduces raw SQL usage.

**Step 2: EF Core vs. EF Framework (EF6)**

|  |  |  |
| --- | --- | --- |
| **Feature** | **EF Core** | **EF6 (Entity Framework Framework)** |
| Platform | Cross-platform (.NET 6/7/8) | Windows-only |
| Lightweight | ✅ | ❌ |
| LINQ Support | ✅ | ✅ |
| Async Queries | ✅ | Partial |
| Compiled Queries | ✅ (better in EF Core 8.0) | ❌ |
| JSON Columns | ✅ | ❌ |
| Performance | Faster | Slower for large apps |

**Step 3: EF Core 8.0 Features**

* **JSON column mapping**: Store structured data like settings directly in a column.
* **Compiled models**: Faster startup for large models.
* **Interceptors**: Hook into EF Core operations for logging, validation, etc.
* **Better bulk operations**: More efficient batch processing.

Code:

dotnet new console -n RetailInventory

cd RetailInventory

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Design

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

Code:

public class Category

{

public int Id { get; set; }public string Name { get; set; }

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

}

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; }

}

public class AppDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

public DbSet<Category> Categories { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(@"Server=ABHRANILMSI\SQLEXPRESS;Database=RetailInventoryDb;Trusted\_Connection=True;TrustServerCertificate=True;");

}

}

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

Code:

dotnet tool install --global dotnet-ef  
dotnet ef migrations add InitialCreate  
dotnet ef database update

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

Code:

class Program

{

static async Task Main(string[] args)

{

await ReadData();

}

static async Task InsertData()

{

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);

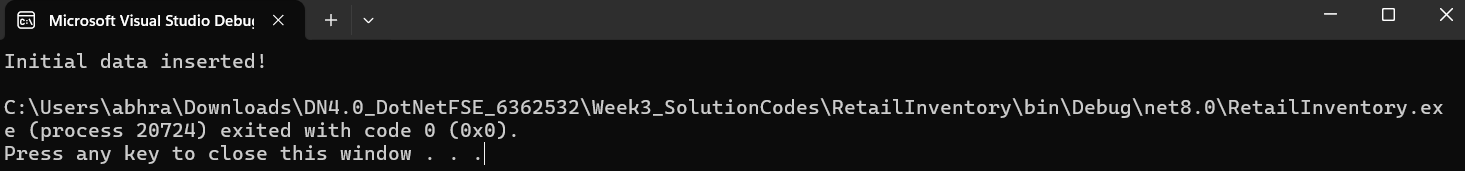
await context.SaveChangesAsync();

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

}

}

Output:



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

Code:

class Program

{

static async Task Main(string[] args)

{

await ReadData();

}

static async Task ReadData()

{

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 by ID (1):");

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

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

Console.WriteLine("\nFirst Product over ₹50,000:");

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

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

}

}

Output:

