**ENTITY FRAMEWORK CORE 8.0**

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

**1. What is ORM?**

ORM (Object-Relational Mapping) is a programming technique that:

- Maps C# classes to database tables

- Maps class properties to table columns

- Maps class instances to table rows

- Handles the conversion between object-oriented programming and relational databases

**How ORM maps C# classes to database tables:**

- Each C# class becomes a table (e.g., `Product` class → `Products` table)

- Class properties become columns (e.g., `Product.Name` → `Products.Name` column)

- Relationships between classes become foreign keys (e.g., `Product.Category` → `CategoryId` foreign key)

**Benefits**:

- Productivity: Write C# code instead of SQL

- Maintainability: Changes in one place affect both code and database

-Abstraction: Work with objects without worrying about database specifics

- Type safety: Compile-time checking of queries

- Database independence: Easier to switch databases

**2. EF Core vs EF Framework**

Entity Framework Core (EF Core):

- Cross-platform (Windows, Linux, macOS)

- Lightweight and modular

- Supports modern features:

- LINQ queries

- Async operations

- Compiled queries for performance

- Better performance overall

- More flexible for modern application needs

- Open source and actively developed

Entity Framework 6 (EF6):

- Windows-only

- More mature with some advanced features

- Better for legacy applications

- Less flexible for modern development

- No longer receiving major updates

**3. EF Core 8.0 Features**

Key features in EF Core 8.0:

- JSON column mapping: Store and query JSON data in relational columns

- Compiled models: Improved startup performance for large models

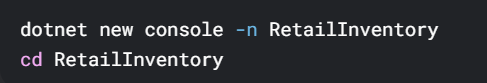
- Interceptors: Middleware for database operations

- Better bulk operations: More efficient updates/deletes

- Enhanced LINQ translation: More queries can be executed in SQL

-Hierarchical data support: Better handling of hierarchical relationships

**4. Creating the .NET Console App**



**5. Installing EF Core Packages**



These packages provide:

- Core EF functionality

- SQL Server database provider

- Design-time tools for migrations

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

**1. Create Models:**



1. **Create AppDbContext:**



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

Local Development Server

"Server=(localdb)\\mssqllocaldb;Database=RetailInventory;Trusted\_Connection=True;"

Full SQL Developer

"Server=localhost;Database=RetailInventory;Trusted\_Connection=True;"

Production developer

"Server=myserver.example.com;Database=RetailInventory;User Id=myuser;Password=mypassword;"

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

1. **Install EF Core CLI :**

dotnet tool install --global dotnet-ef

1. **Create Initial Migration:**

dotnet ef migrations add InitialCreate

1. **Apply Migration to Create Database:**

dotnet ef database update

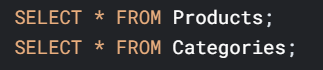
**4.Verify in SQL Server:**

**Using Azure Data Studio:**

**-Connect to your LocalDB instance.**

**-Navigate to the RetailInventory database.**

**-Run query**



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

1. **Insert Data in Program.cs:**

using RetailInventory;

using Microsoft.EntityFrameworkCore;

// Create a new DbContext instance

await using var context = new AppDbContext();

// Check if database already has data

if (!await context.Categories.AnyAsync())

{

// Create categories

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

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

var clothing = new Category { Name = "Clothing" };

// Add categories to context

await context.Categories.AddRangeAsync(electronics, groceries, clothing);

// Create products

var products = new List<Product>

{

new() { Name = "Laptop", Price = 75000, Category = electronics },

new() { Name = "Smartphone", Price = 45000, Category = electronics },

new() { Name = "Rice Bag", Price = 1200, Category = groceries },

new() { Name = "Wheat Flour", Price = 500, Category = groceries },

new() { Name = "T-Shirt", Price = 800, Category = clothing },

new() { Name = "Jeans", Price = 2000, Category = clothing }

};

// Add products to context

await context.Products.AddRangeAsync(products);

// Save all changes to database

var recordsInserted = await context.SaveChangesAsync();

Console.WriteLine($"Successfully inserted {recordsInserted} records.");

}

else

{

Console.WriteLine("Database already contains data - no seeding required.");

}

**2. Run the App:**

Dotnet run

3.**Verify in SQL Server:**

Using SQL Server Management Studio (SSMS):

Connect to (localdb)\mssqllocaldb

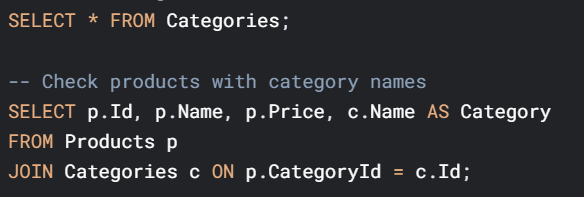
Open the RetailInventory database

Right-click each table and select "Select Top 1000 Rows"

Verify you see:

Categories: Electronics, Groceries, Clothing

Products with correct category associations



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

1. **Retrieve All Products:**

try

{

Console.WriteLine("=== All Products ===");

// Retrieve all products asynchronously

var products = await context.Products

.Include(p => p.Category) // Include related category data

.OrderBy(p => p.Name) // Sort alphabetically

.ToListAsync();

if (products.Any())

{

foreach (var p in products)

{

Console.WriteLine($"{p.Id}. {p.Name} - ₹{p.Price:N0} ({p.Category.Name})");

}

Console.WriteLine($"Total: {products.Count} products");

}

else

{

Console.WriteLine("No products found in inventory.");

}

}

catch (Exception ex)

{

Console.WriteLine($"Error retrieving products: {ex.Message}");

}

1. **Find by ID:**

Console.WriteLine("\n=== Find Product by ID ===");

Console.Write("Enter product ID to search: ");

if (int.TryParse(Console.ReadLine(), out int searchId))

{

var product = await context.Products

.Include(p => p.Category)

.FirstOrDefaultAsync(p => p.Id == searchId);

if (product != null)

{

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

Console.WriteLine($"Price: ₹{product.Price:N0}");

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

}

else

{

Console.WriteLine($"No product found with ID {searchId}");

}

}

else

{

Console.WriteLine("Invalid ID entered. Please enter a number.");

}

**3. FirstOrDefault with Condition:**

Console.WriteLine("\n=== Find Expensive Products ===");

var priceThreshold = 50000;

var expensiveProducts = await context.Products

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

.Include(p => p.Category)

.OrderByDescending(p => p.Price)

.ToListAsync();

Console.WriteLine($"Products above ₹{priceThreshold:N0}:");

if (expensiveProducts.Any())

{

foreach (var p in expensiveProducts)

{

Console.WriteLine($"{p.Name} - ₹{p.Price:N0} ({p.Category.Name})");

}

}

else

{

Console.WriteLine($"No products found above ₹{priceThreshold:N0}");

}

**ANALYSIS:**

This analysis examines the complete implementation of the retail inventory system using Entity Framework Core, covering architectural decisions, implementation quality, and potential improvements.

1. Architectural Overview

ORM Implementation

Effective Object-Relational Mapping: The system properly maps C# classes (Product, Category) to database tables with:

Clear primary key definitions (Id properties)

Well-configured relationships (Category→Products one-to-many)

Appropriate navigation properties in both directions

Layering Structure

The solution follows a simple but effective two-layer architecture:

Data Layer: AppDbContext and entity models

Application Layer: Console application with business logic

2. Strengths

Database Management

Migrations: Proper use of EF Core migrations for schema evolution

Connection Handling: Correct implementation of DbContext with connection string configuration

Async Operations: Consistent use of async/await pattern (AddAsync, SaveChangesAsync)

Data Access Patterns

Flexible Querying: Demonstrated multiple query approaches:

ToListAsync() for full collections

FindAsync() for primary key lookups

FirstOrDefaultAsync() with predicates for conditional queries

Eager Loading: Proper use of Include() for related data

Code Quality

Error Handling: Basic exception handling in data retrieval

Validation: Input validation for ID lookup

Organization: Logical separation of concerns in different labs