# 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?

**Object-Relational Mapping (ORM)** is a programming technique that allows developers to interact with databases using object-oriented programming languages instead of writing raw SQL queries.

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

* C# classes represent database tables
* Properties in classes represent columns in tables
* Objects (instances) represent rows in tables
* Relationships between classes represent foreign key relationships

**Benefits:**

* **Productivity**: Write less code, focus on business logic instead of SQL
* **Maintainability**: Changes to database schema can be managed through code
* **Abstraction from SQL**: Developers can work with familiar object-oriented concepts

#### 2. EF Core vs EF Framework:

**EF Core:**

* Cross-platform (Windows, Linux, macOS)
* Lightweight and modular
* Supports modern features like LINQ, async queries, and compiled queries
* Better performance
* Actively developed with new features

**EF Framework (EF6):**

* Windows-only
* More mature with extensive features
* Less flexible
* Legacy support, minimal new development

#### 3. EF Core 8.0 Features:

* **JSON column mapping**: Store and query JSON data directly in database columns
* **Improved performance with compiled models**: Pre-compiled queries for better performance
* **Interceptors**: Hook into EF Core operations for logging, caching, etc.
* **Better bulk operations**: Improved performance for large data operations

#### 4. Create a .NET Console App:

dotnet new console -n RetailInventory

cd RetailInventory

#### 5. Install EF Core Packages:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Design

## 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:

Create a new file: Models/Category.cs

using System.Collections.Generic;

namespace RetailInventory.Models

{

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

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

}

}

Create a new file: Models/Product.cs

namespace RetailInventory.Models

{

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; }

}

}

#### 2. Create AppDbContext:

Create a new file: Data/AppDbContext.cs

using Microsoft.EntityFrameworkCore;

using RetailInventory.Models;

namespace RetailInventory.Data

{

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=.;Database=RetailInventoryDB;Trusted\_Connection=true;TrustServerCertificate=true;");

}

}

}

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

{

"ConnectionStrings": {

"DefaultConnection": "Server=.;Database=RetailInventoryDB;Trusted\_Connection=true;TrustServerCertificate=true;"

}

}

## 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

#### 2. Create Initial Migration:

dotnet ef migrations add InitialCreate

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

**Generated Migration File Example:**

using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace RetailInventory.Migrations

{

/// <inheritdoc />

public partial class InitialCreate : Migration

{

/// <inheritdoc />

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Categories",

columns: table => new

{

Id = table.Column<int>(type: "int", nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Name = table.Column<string>(type: "nvarchar(max)", nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Categories", x => x.Id);

});

migrationBuilder.CreateTable(

name: "Products",

columns: table => new

{

Id = table.Column<int>(type: "int", nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Name = table.Column<string>(type: "nvarchar(max)", nullable: false),

Price = table.Column<decimal>(type: "decimal(18,2)", nullable: false),

CategoryId = table.Column<int>(type: "int", nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Products", x => x.Id);

table.ForeignKey(

name: "FK\_Products\_Categories\_CategoryId",

column: x => x.CategoryId,

principalTable: "Categories",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

});

migrationBuilder.CreateIndex(

name: "IX\_Products\_CategoryId",

table: "Products",

column: "CategoryId");

}

/// <inheritdoc />

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(

name: "Products");

migrationBuilder.DropTable(

name: "Categories");

}

}

}

#### 3. Apply Migration to Create Database:

dotnet ef database update

#### 4. Verify in SQL Server:

Open SQL Server Management Studio (SSMS) or Azure Data Studio and confirm that tables Products and Categories are created.

## 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:

Update your Program.cs file:

using Microsoft.EntityFrameworkCore;

using RetailInventory.Data;

using RetailInventory.Models;

Console.WriteLine("Starting Retail Inventory System...");

using var context = new AppDbContext();

// Create categories

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

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

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

// Create products

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);

// Save changes to database

await context.SaveChangesAsync();

Console.WriteLine("Data inserted successfully!");

Console.WriteLine($"Electronics Category ID: {electronics.Id}");

Console.WriteLine($"Groceries Category ID: {groceries.Id}");

Console.WriteLine($"Laptop Product ID: {product1.Id}");

Console.WriteLine($"Rice Bag Product ID: {product2.Id}");

#### 2. Run the App:

dotnet run

#### 3. Verify in SQL Server:

Check that the data is inserted correctly using SQL Server Management Studio or Azure Data Studio.

**Expected Output:**

Starting Retail Inventory System...

Data inserted successfully!

Electronics Category ID: 1

Groceries Category ID: 2

Laptop Product ID: 1

Rice Bag Product ID: 2

## 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. Complete Program.cs for Data Retrieval:

Update your Program.cs file:

using Microsoft.EntityFrameworkCore;

using RetailInventory.Data;

using RetailInventory.Models;

Console.WriteLine("Starting Retail Inventory System...");

using var context = new AppDbContext();

// Check if data already exists

var existingCategories = await context.Categories.CountAsync();

if (existingCategories == 0)

{

Console.WriteLine("Inserting initial data...");

// Create categories

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

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

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

// Create products

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);

// Save changes to database

await context.SaveChangesAsync();

Console.WriteLine("Data inserted successfully!");

}

Console.WriteLine("\n--- Retrieving Data ---");

// 1. Retrieve All Products

Console.WriteLine("\n1. All Products:");

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

foreach (var p in products)

{

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

}

// 2. Find by ID

Console.WriteLine("\n2. Find Product by ID (1):");

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

Console.WriteLine($"Found: {product?.Name ?? "Not found"}");

// 3. FirstOrDefault with Condition

Console.WriteLine("\n3. Find Expensive Product (Price > 50000):");

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

Console.WriteLine($"Expensive: {expensive?.Name ?? "Not found"}");

Console.WriteLine("\nData retrieval completed!");

#### Expected Output:

Starting Retail Inventory System...

--- Retrieving Data ---

1. All Products:

Laptop - ₹75000

Rice Bag - ₹1200

2. Find Product by ID (1):

Found: Laptop

3. Find Expensive Product (Price > 50000):

Expensive: Laptop

Data retrieval completed!

**Lab 6: Updating and Deleting Records**

### Scenario:

The store updates product prices and removes discontinued items.

### Objective:

Update and delete records using EF Core.

### Steps:

#### 1. Complete Program.cs for Update and Delete Operations:

Update your Program.cs file:

using Microsoft.EntityFrameworkCore;

using RetailInventory.Data;

using RetailInventory.Models;

Console.WriteLine("Starting Retail Inventory System...");

using var context = new AppDbContext();

// Check if data already exists

var existingCategories = await context.Categories.CountAsync();

if (existingCategories == 0)

{

Console.WriteLine("Inserting initial data...");

// Create categories

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

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

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

// Create products

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);

// Save changes to database

await context.SaveChangesAsync();

Console.WriteLine("Data inserted successfully!");

}

Console.WriteLine("\n--- Before Updates ---");

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

foreach (var p in allProducts)

{

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

}

Console.WriteLine("\n--- Updating and Deleting Records ---");

// 1. Update a Product

Console.WriteLine("\n1. Updating Laptop price from ₹75000 to ₹70000...");

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

if (productToUpdate != null)

{

productToUpdate.Price = 70000;

await context.SaveChangesAsync();

Console.WriteLine("Laptop price updated successfully!");

}

else

{

Console.WriteLine("Laptop not found for update.");

}

// 2. Delete a Product

Console.WriteLine("\n2. Deleting Rice Bag...");

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

if (toDelete != null)

{

context.Products.Remove(toDelete);

await context.SaveChangesAsync();

Console.WriteLine("Rice Bag deleted successfully!");

}

else

{

Console.WriteLine("Rice Bag not found for deletion.");

}

Console.WriteLine("\n--- After Updates ---");

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

foreach (var p in updatedProducts)

{

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

}

Console.WriteLine("\nUpdate and delete operations completed!");

#### Expected Output:

Starting Retail Inventory System...

--- Before Updates ---

Laptop - ₹75000

Rice Bag - ₹1200

--- Updating and Deleting Records ---

1. Updating Laptop price from ₹75000 to ₹70000...

Laptop price updated successfully!

2. Deleting Rice Bag...

Rice Bag deleted successfully!

--- After Updates ---

Laptop - ₹70000

Update and delete operations completed!