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?**
    - Explain how ORM maps C# classes to database tables.
    - Benefits: Productivity, maintainability, and abstraction from SQL.
2. **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.
3. **EF Core 8.0 Features:**
    - JSON column mapping.
    - Improved performance with compiled models.
    - Interceptors and better bulk 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:**

```
public class Category {
   public int Id { get; set; }
   public string Name { get; set; }
   public List 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; }
}
```

2. **Create AppDbContext:**

```
public class AppDbContext : DbContext {
   public DbSet Products { get; set; }
   public DbSet Categories { get; set; }
   protected override void OnConfiguring(DbContextOptionsBuilder optionsBuild
er) {
      optionsBuilder.UseSqlServer("Your_Connection_String_Here");
   }
}
```

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

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.

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:**

```
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 = electro
nics };
var product2 = new Product { Name = "Rice Bag", Price = 1200, Category = groceri
es };

await context.Products.AddRangeAsync(product1, product2);
await context.SaveChangesAsync();
```

2. **Run the App:**
```
dotnet run
```
3. **Verify in SQL Server:**
   Check that the data is inserted correctly.

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. **Retrieve All Products:**

```
var products = await context.Products.ToListAsync();
foreach (var p in products)
  Console.WriteLine($"{p.Name} - ₹{p.Price}");
```

2. **Find by ID:**

```
var product = await context.Products.FindAsync(1);
Console.WriteLine($"Found: {product?.Name}");
```

3. **FirstOrDefault with Condition:**

```
var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 5000
0);
Console.WriteLine($"Expensive: {expensive?.Name}");
```

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. **Update a Product:**

```
var product = await context.Products.FirstOrDefaultAsync(p => p.Name == "Lapt
op");
if (product != null) {
   product.Price = 70000;
   await context.SaveChangesAsync();
}
```

2. **Delete a Product:**

```
var toDelete = await context.Products.FirstOrDefaultAsync(p => p.Name == "Rice
 Bag");
if (toDelete != null) {
   context.Products.Remove(toDelete);
   await context.SaveChangesAsync();
}
```

Lab 7: Writing Queries with LINQ

Scenario:
The store wants to filter and sort products for reporting.

Objective:
Use Where, Select, OrderBy, and project into DTOs.

Steps:

1. **Filter and Sort:**

```
var filtered = await context.Products
  .Where(p => p.Price > 1000)
  .OrderByDescending(p => p.Price)
  .ToListAsync();
```

2. **Project into DTO:**

```
var productDTOs = await context.Products
  .Select(p => new { p.Name, p.Price })
  .ToListAsync();
```

Lab 8: Managing Migrations and Schema Changes

Scenario:
The store wants to add a new field StockQuantity to track inventory levels for each product.

Objective:
Learn how to update the schema using EF Core migrations.

Steps:

1. **Update the Product Model:**

```
public int StockQuantity { get; set; }
```

2. **Create a New Migration:**

```
dotnet ef migrations add AddStockQuantity
```

3. **Apply the Migration:**

```
dotnet ef database update
```

4. **Verify in SQL Server:**
   Confirm that the StockQuantity column is added.

Lab 9: Seeding Data During Migrations

Scenario:
The store wants to pre-load some categories and products when the database is created.

Objective:
Use HasData() to seed data during migrations.

Steps:

1. **Modify OnModelCreating:**

```
modelBuilder.Entity().HasData(
  new Category { Id = 1, Name = "Electronics" },
  new Category { Id = 2, Name = "Groceries" }
);
modelBuilder.Entity().HasData(
  new Product { Id = 1, Name = "Smartphone", Price = 25000, CategoryId = 1, Sto
ckQuantity = 50 },
  new Product { Id = 2, Name = "Wheat Flour", Price = 800, CategoryId = 2, Stock
Quantity = 100 }
);
```

2. **Create and Apply Migration:**

```
dotnet ef migrations add SeedInitialData
dotnet ef database update
```

Lab 10: Eager, Lazy, and Explicit Loading

Scenario:
The store dashboard needs to show products along with their categories.

Objective:
Understand different loading strategies for related data.

Steps:

1. **Eager Loading:**

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

2. **Explicit Loading:**

```
var product = await context.Products.FirstAsync();
await context.Entry(product).Reference(p => p.Category).LoadAsync();
```

3. **Lazy Loading (Optional):**
    - Install: Microsoft.EntityFrameworkCore.Proxies
    - Enable in OnConfiguring:

```
optionsBuilder.UseLazyLoadingProxies().UseSqlServer(...);
```

    - Mark navigation properties as virtual

Lab 11: Configuring One-to-One and Many-to-Many Relationships

Scenario:
The store wants to track product details (e.g., warranty info) and allow products to belong to multiple tags (e.g., “On Sale”, “New Arrival”).

Objective:
Configure one-to-one and many-to-many relationships.

Steps:

1. **One-to-One:**

```
public class ProductDetail {
  public int ProductDetailId { get; set; }
  public string WarrantyInfo { get; set; }
  public int ProductId { get; set; }
  public Product Product { get; set; }
}
modelBuilder.Entity()
  .HasOne(p => p.ProductDetail)
  .WithOne(pd => pd.Product)
  .HasForeignKey(pd => pd.ProductId);
```

2. **Many-to-Many:**

```
public class Tag {
  public int Id { get; set; }
  public string Name { get; set; }
  public List Products { get; set; }
}
public class Product {
  ...
```

```
    public List Tags { get; set; }
}
```

EF Core 8 handles many-to-many automatically.

Lab 12: Navigating Circular References

Scenario:
The store's API returns product and category data, but circular references cause serialization issues.

Objective:
Handle circular references in navigation properties.

Steps:

1. **Use DTOs for API Responses:**

```
public class ProductDTO {
    public string Name { get; set; }
    public string CategoryName { get; set; }
}
```

2. **Project to DTO:**

```
var productDTOs = await context.Products
    .Select(p => new ProductDTO {
        Name = p.Name,
        CategoryName = p.Category.Name
    }).ToListAsync();
```

3. **Alternative:** Use [JsonIgnore] on navigation properties.

Lab 13: Query Caching and Tracking Behavior

Scenario:
The store runs frequent read-only reports and wants to optimize performance.

Objective:
Use AsNoTracking and compiled queries.

Steps:

1. **AsNoTracking:**

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

2. **Compiled Query:**

```
static readonly Func<appdbcontext, decimal, task<list>> _expensiveProducts =<
/appdbcontext, decimal, task<list
  EF.CompileAsyncQuery((AppDbContext ctx, decimal price) =>
    ctx.Products.Where(p => p.Price > price));
var result = await _expensiveProducts(context, 10000);
```

Lab 14: Batch Processing and Bulk Operations

Scenario:
The store wants to update stock levels for 1000+ products after a stock audit.

Objective:
Use batch operations for performance.

Steps:

1. **Install EFCore.BulkExtensions:**

```
dotnet add package EFCore.BulkExtensions
```

2. **Use Bulk Update:**

```
await context.BulkUpdateAsync(productList);
```

3. **Compare with regular SaveChangesAsync() for performance.

Lab 15: Handling Concurrency with RowVersion

Scenario:
Two employees try to update the same product's stock at the same time.

Objective:
Use RowVersion to detect and handle concurrency conflicts.

Steps:

1. **Add RowVersion to Product:**

```
[Timestamp]
public byte[] RowVersion { get; set; }
```

2. **Handle Concurrency Exception:**

```
try {
   await context.SaveChangesAsync();
} catch (DbUpdateConcurrencyException ex) {
   Console.WriteLine("Concurrency conflict detected.");
}
```