**EF Core 8.0 Guided Mandatory\_Hands\_On Exercises**

**Submitted By:** Gyarina Rauniyar  
**Lab1: Understanding ORM with a Retail Inventory Syste****m**

## **Scenario:**

I am 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:**

Understanding what ORM is and how EF Core helps bridge the gap between C# objects and relational tables.

## **Task Solutions**

### **1. What is ORM?**

ORM (Object-Relational Mapping) allows developers to map C# classes to relational database tables. It enables data manipulation using C# code instead of raw SQL queries.

**Example Mapping:**

public class Product  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public int Stock { get; set; }  
}

In this example, EF Core will automatically map the Product class to a database table Products.

**Benefits:**

* Increased productivity
* Better maintainability
* Reduced need for manual SQL
* Database abstraction layer

### **2. EF Core vs EF Framework**

| Feature | EF Core 8.0 | EF Framework (EF6) |
| --- | --- | --- |
| Platform | Cross-platform | Windows-only |
| Lightweight | Yes | Heavier, mature |
| Modern Features | LINQ, Async, Compiled Queries, JSON columns | Limited modern features |
| Flexibility | High | Less flexible |

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

* JSON column mapping
* Compiled models for improved performance
* Interceptors and better bulk operations

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

Commands used:

dotnet new console -n RetailInventory  
cd RetailInventory

### **5. Install EF Core Packages**

Commands used:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer  
dotnet add package Microsoft.EntityFrameworkCore.Design

### **6. C# Code for the Retail Inventory System**

**Product.cs**

namespace RetailInventory  
{  
 public class Product  
 {  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public int Stock { get; set; }  
 public int CategoryId { get; set; }  
 public Category Category { get; set; }  
 }  
}

**Category.cs**

namespace RetailInventory  
{  
 public class Category  
 {  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public List<Product> Products { get; set; }  
 }  
}

**InventoryContext.cs**

using Microsoft.EntityFrameworkCore;  
  
namespace RetailInventory  
{  
 public class InventoryContext : DbContext  
 {  
 public DbSet<Product> Products { get; set; }  
 public DbSet<Category> Categories { get; set; }  
  
 protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  
 {  
 optionsBuilder.UseSqlServer(@"Server=localhost\\SQLEXPRESS;Database=RetailInventoryDB;Trusted\_Connection=True;TrustServerCertificate=True;");  
 }  
 }  
}

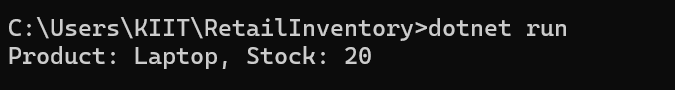
**Program.cs**

using System;  
using System.Linq;  
  
namespace RetailInventory  
{  
 class Program  
 {  
 static void Main(string[] args)  
 {  
 using (var context = new InventoryContext())  
 {  
 // Add Category  
 var category = new Category { Name = "Electronics" };  
 context.Categories.Add(category);  
 context.SaveChanges();  
  
 // Add Product  
 var product = new Product  
 {  
 Name = "Laptop",  
 Stock = 20,  
 CategoryId = category.Id  
 };  
 context.Products.Add(product);  
 context.SaveChanges();  
  
 // Display Products  
 var products = context.Products.ToList();  
 foreach (var p in products)  
 {  
 Console.WriteLine($"Product: {p.Name}, Stock: {p.Stock}");  
 }  
 }  
 }  
 }  
}

### **7. Migration and Database Commands**

dotnet ef migrations add InitialCreate  
dotnet ef database update  
dotnet run

### **8. Output Screenshot**

Product: Laptop, Stock: 20

## 

## **Conclusion**

The lab demonstrates how EF Core bridges the gap between C# code and relational databases using ORM principles. I successfully created a console application, mapped C# classes to database tables, performed database operations, and viewed the output.

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

## **Scenario:**

The retail store wants to store product and category data in a SQL Server database.

## **Objective:**

Configuring DbContext and connecting to SQL Server using C# and EF Core.

## **Task Solutions**

### **1. Project Setup**

Created a new .NET Console project using the following commands:

dotnet new console -n RetailStoreContextLab   
cd RetailStoreContextLab

Installed EF Core packages:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer   
dotnet add package Microsoft.EntityFrameworkCore.Design

### **2. C# Model Classes**

**Category.cs**

public class Category   
{   
 public int Id { get; set; }   
 public string Name { get; set; }   
 public List<Product> Products { get; set; }   
}

**Product.cs**

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; }   
}

### **3. Database Context Configuration**

**AppDbContext.cs**

using Microsoft.EntityFrameworkCore;   
  
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=localhost\SQLEXPRESS;Database=RetailInventoryDB\_Lab2;Trusted\_Connection=True;TrustServerCertificate=True;");   
 }   
}

(Note: A different database name RetailInventoryDB\_Lab2 was used to avoid conflicts with previous labs.)

### **4. Testing the Database Context in Program.cs**

using System;   
using System.Linq;   
  
class Program   
{   
 static void Main(string[] args)   
 {   
 using (var context = new AppDbContext())   
 {   
 // Add Category   
 var category = new Category { Name = "Appliances" };   
 context.Categories.Add(category);   
 context.SaveChanges();   
  
 // Add Product   
 var product = new Product   
 {   
 Name = "Washing Machine",   
 Price = 35000,   
 CategoryId = category.Id   
 };   
 context.Products.Add(product);   
 context.SaveChanges();   
  
 // Display Products   
 var products = context.Products.ToList();   
 foreach (var p in products)   
 {   
 Console.WriteLine($"Product: {p.Name}, Price: {p.Price}");   
 }   
 }   
 }   
}

### **5. Migration and Database Creation Commands**

dotnet ef migrations add InitialCreate   
dotnet ef database update   
dotnet run

### **6. Output Screenshot**

Product: Washing Machine, Price: 35000 Screenshot 2025-07-06 115703

## **Conclusion**

The lab demonstrates successful creation of model classes, configuration of AppDbContext, and interaction with a SQL Server database using C#. The output confirms that the database is properly set up and working.

**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 I’ve defined. I’ll use EF Core CLI to generate and apply migrations.

## **Objective:**

Learning how to use EF Core CLI to manage database schema changes.

## **Task Solutions**

### **1. EF Core CLI Installation**

Installed EF Core CLI globally using the following command:

dotnet tool install --global dotnet-ef

### **2. Created Initial Migration**

Navigated to the project folder:

cd RetailStoreContextLab

Created an initial migration named InitialCreateLab3 to avoid conflicts with previous migrations:

dotnet ef migrations add InitialCreateLab3

A **Migrations** folder was generated containing code that represents the database schema based on the C# models.

### **3. Applied Migration to Create the Database**

Executed the following command to apply the migration and create the database:

dotnet ef database update

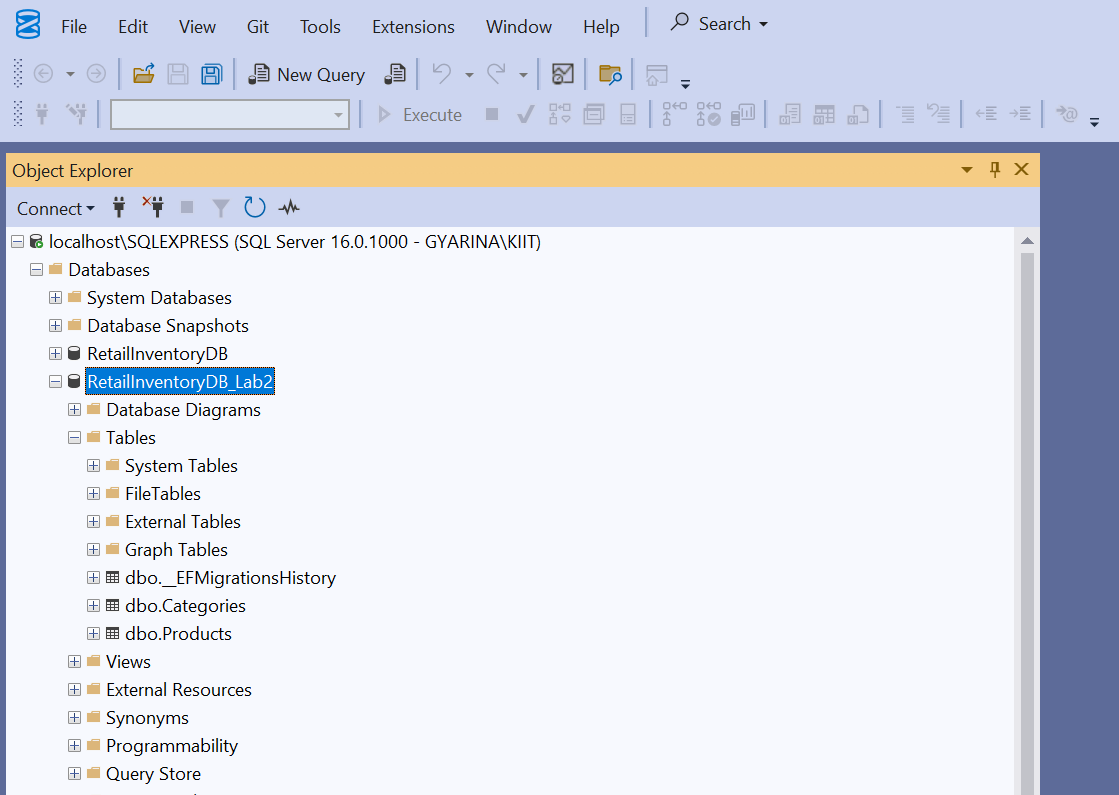
The database RetailInventoryDB\_Lab2 was successfully created in SQL Server.

### **4. Verified Database in SQL Server Management Studio (SSMS)**

Steps followed:

* Opened **SQL Server Management Studio (SSMS)**
* Connected to server: localhost\SQLEXPRESS
* Expanded Databases → RetailInventoryDB\_Lab2
* Expanded Tables

Confirmed that the following tables were created: ✅ dbo.Categories  
✅ dbo.Products



This confirmed that the EF Core migration and database update were successful.

## **Conclusion**

Successfully used EF Core CLI to generate and apply migrations. Verified that the database and tables were created as expected in SQL Server. The process demonstrated how EF Core bridges the gap between C# models and the relational database structure.

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

## **Scenario:**

The store manager wants to add initial product categories and products to the system.

## **Objective:**

Using EF Core to insert records using AddRangeAsync and SaveChangesAsync.

## **Task Solutions**

### **1. Program.cs Code to Insert Data**

The following C# code was written in Program.cs to insert initial data:

using System;   
using System.Threading.Tasks;   
  
class Program   
{   
 static async Task Main(string[] args)   
 {   
 using var context = new AppDbContext();   
  
 // Insert Categories   
 var electronics = new Category { Name = "Electronics" };   
 var groceries = new Category { Name = "Groceries" };   
 await context.Categories.AddRangeAsync(electronics, groceries);   
  
 // Insert Products linked to Categories   
 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 the database   
 await context.SaveChangesAsync();   
  
 Console.WriteLine("Initial data inserted successfully!");   
 }   
}

### **2. Ran the Application**

Executed the following command in the project directory:

dotnet run

Output:

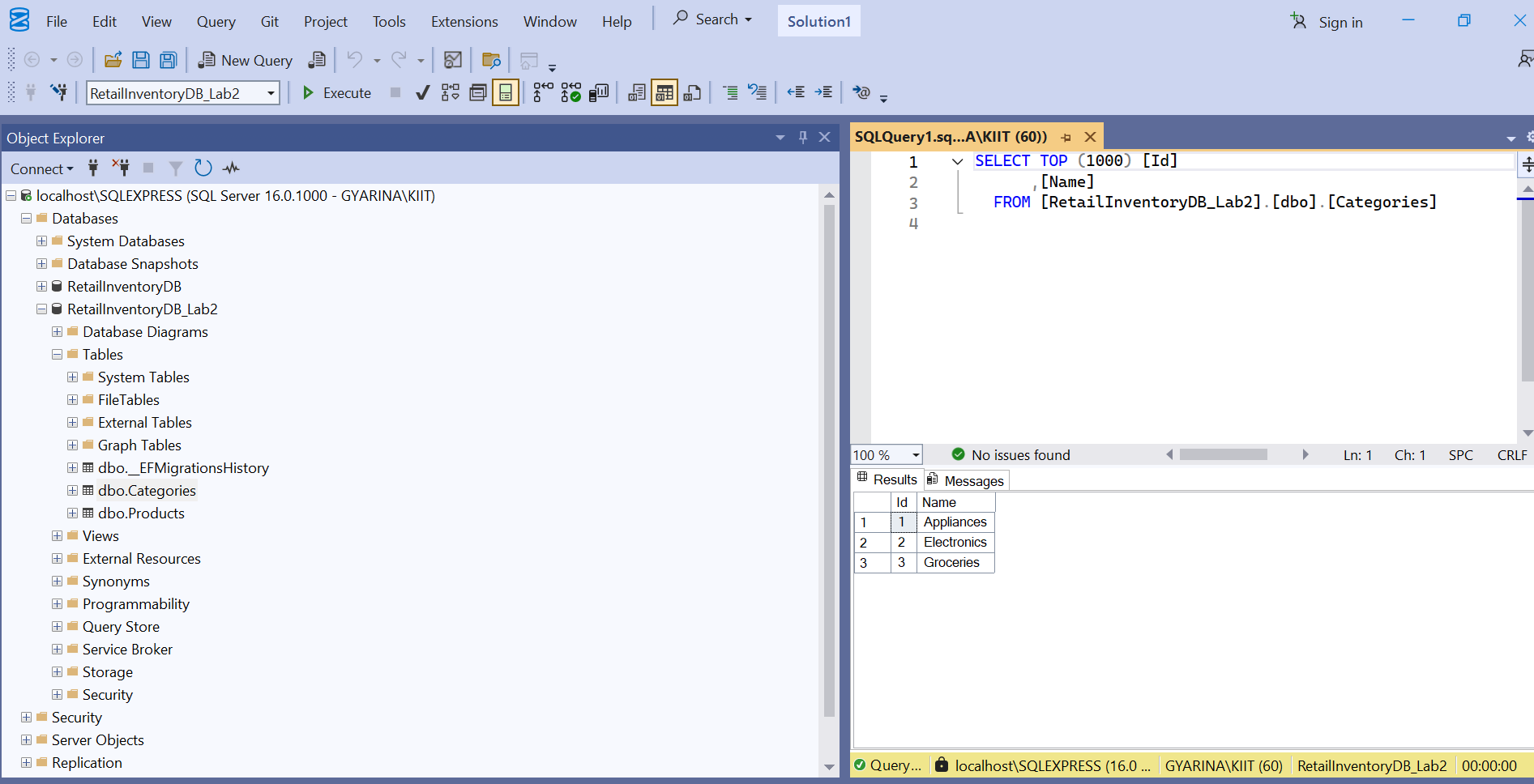
Initial data inserted successfully!

Screenshot 2025-07-06 160545

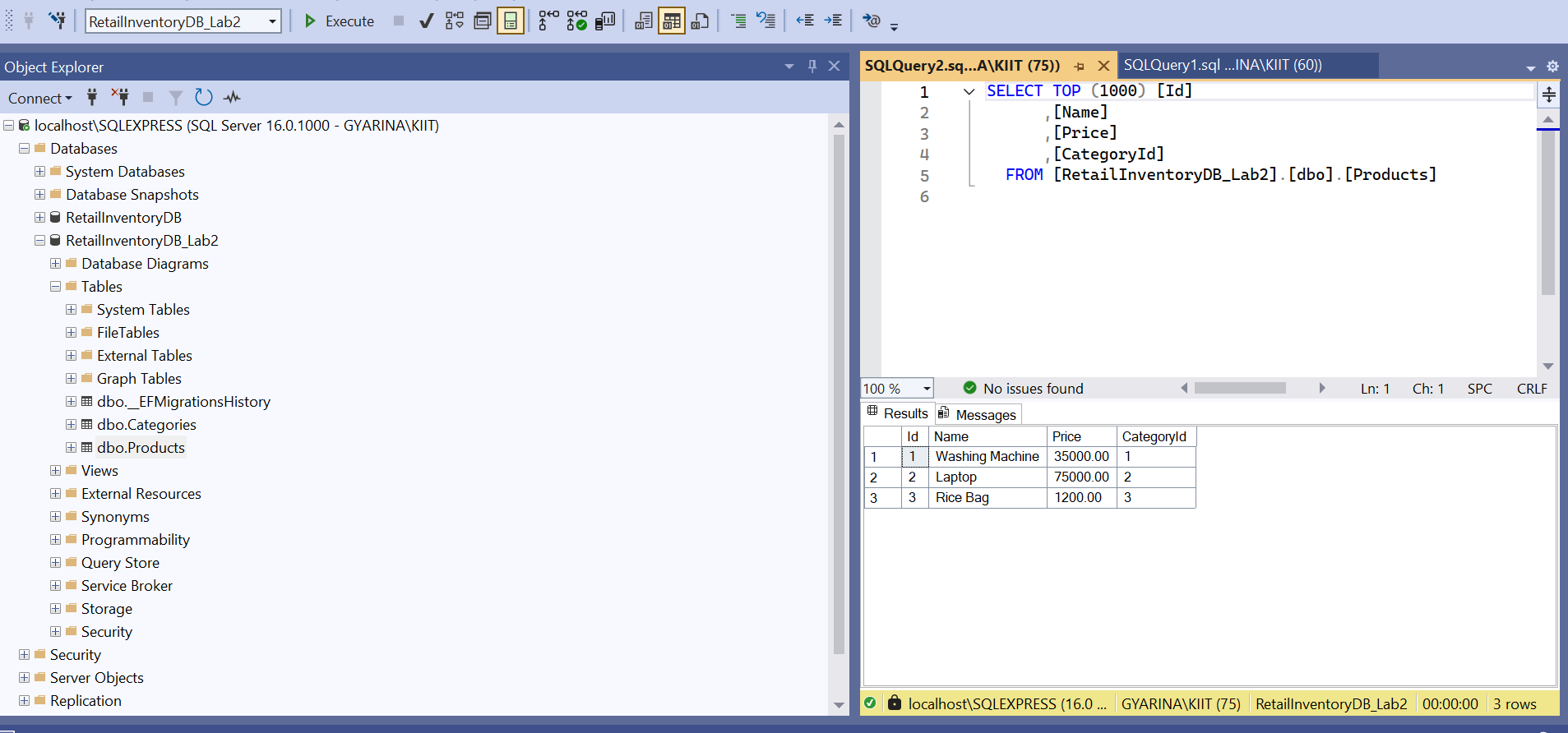
### **3. Verified Data in SQL Server Management Studio (SSMS)**

Steps followed:

* Opened **SSMS**
* Connected to server: localhost\SQLEXPRESS
* Expanded Databases → RetailInventoryDB\_Lab2
* Expanded Tables
* Right-clicked **dbo.Categories** → Selected **Select Top 1000 Rows**



* Right-clicked **dbo.Products** → Selected **Select Top 1000 Rows**



**Verified Records:**

* Categories:
  + Electronics
  + Groceries
* Products:
  + Laptop (linked to Electronics)
  + Rice Bag (linked to Groceries)

## **Conclusion**

Successfully inserted initial product categories and products into the database using EF Core’s AddRangeAsync and SaveChangesAsync. Verified that the data is present in SQL Server, confirming the lab objective is achieved.

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

## **Scenario:**

The store wants to display product details on the dashboard.

## **Objective:**

Using FindAsync, FirstOrDefaultAsync, and ToListAsync to retrieve data from the database using EF Core.

## **Task Solutions**

### **1. Program.cs Code to Retrieve Data**

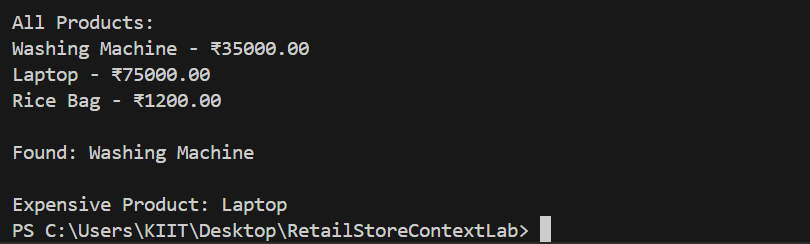
The following C# code was written in Program.cs to retrieve data:

using System;   
using System.Linq;   
using System.Threading.Tasks;   
using Microsoft.EntityFrameworkCore;   
  
class Program   
{   
 static async Task Main(string[] args)   
 {   
 using var context = new AppDbContext();   
  
 // 1. Retrieve All Products   
 var products = await context.Products.ToListAsync();   
 Console.WriteLine("\nAll Products:");   
 foreach (var p in products)   
 Console.WriteLine($"{p.Name} - ₹{p.Price:0.00}");   
  
 // 2. Find by ID   
 var product = await context.Products.FindAsync(1);   
 Console.WriteLine($"\nFound: {product?.Name}");   
  
 // 3. FirstOrDefault with Condition   
 var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 50000);   
 Console.WriteLine($"\nExpensive Product: {expensive?.Name}");   
 }   
}

### **2. Ran the Application**

Executed the following command in the project directory:

dotnet run

**Output Observed:**

### **3. Explanation of Output**

* All products from the database were displayed using ToListAsync()
* The product with ID = 1 (Washing Machine) was found using FindAsync(1)
* The first product with price greater than ₹50,000 (Laptop) was found using FirstOrDefaultAsync

## **Conclusion**

Successfully retrieved product details from the database using EF Core’s ToListAsync, FindAsync, and FirstOrDefaultAsync methods. Verified that the correct data was displayed on the console, fulfilling the lab objectives.