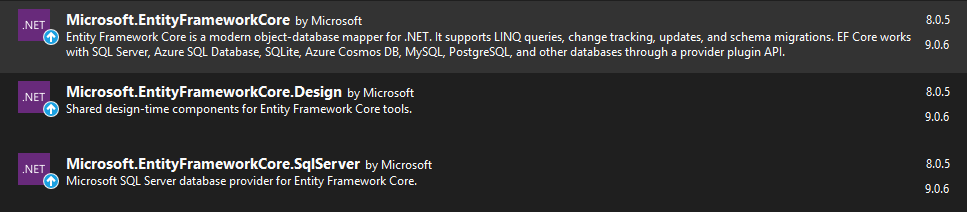
**Lab 1: Understanding ORM with a Retail Inventory System**

### 1. What is ORM and how does it map C# classes to database tables?

ORM (Object-Relational Mapping) allows developers to work with databases using C# objects instead of SQL. In this approach, C# classes map to database tables, and class properties map to table columns. Relationships like one-to-many are represented using navigation properties, making data access more intuitive and object-oriented.

### 2. What are the benefits of using ORM?

ORM improves productivity by reducing the need for SQL, enhances maintainability through features like migrations, and provides abstraction, allowing developers to focus on business logic rather than database syntax.



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

**CODE:**

**# Category.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RetailInventory.Models

{

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

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

}

}

**#Product.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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; }

}

}

**#AppDbContext.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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=SOHAM\_09\\SQLEXPRESS;Database=RetailDb;Trusted\_Connection=True;TrustServerCertificate=True;");

}

}

}

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

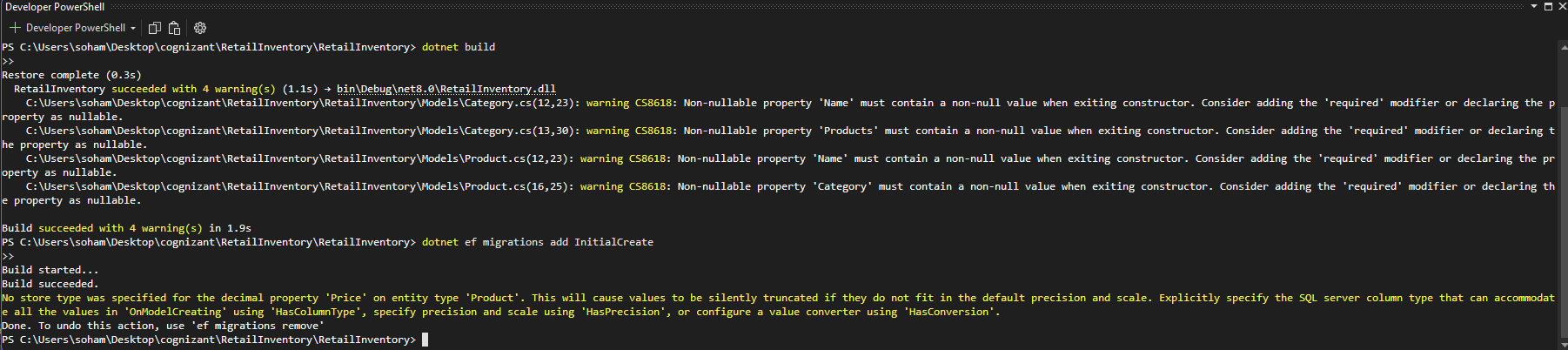
**CODE(bash):**

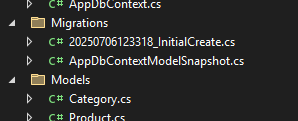
dotnet tool install --global dotnet-ef

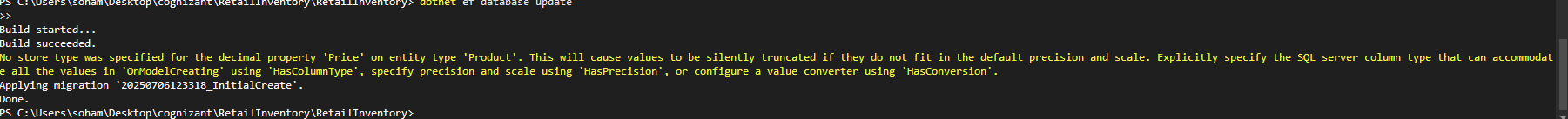
dotnet ef migrations add InitialCreate

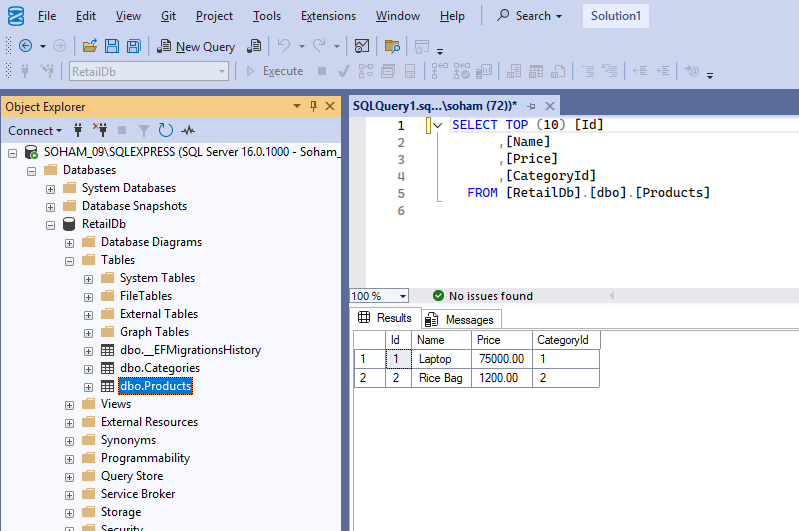
dotnet ef database update

**OUTPUT:**

****



****

****

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

**CODE:**

using RetailInventory.Data;

using RetailInventory.Models;

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 };

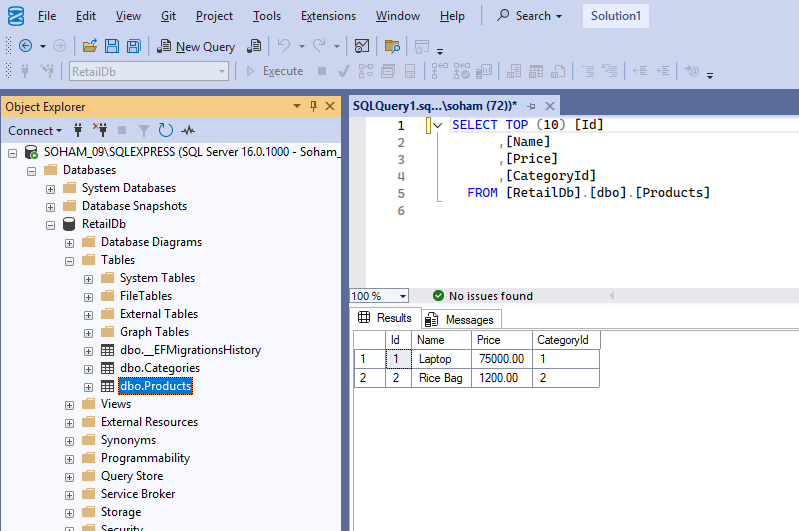
var product2 = new Product { Name = "Rice Bag", Price = 1200, Category = groceries };

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

await context.SaveChangesAsync();

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

**OUTPUT:**

****

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

**CODE:**

using RetailInventory.Data;

using RetailInventory.Models;

using Microsoft.EntityFrameworkCore;

var context = new AppDbContext();

// 1. Get All Products

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

foreach (var p in products)

{

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

}

// 2. Find by ID

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

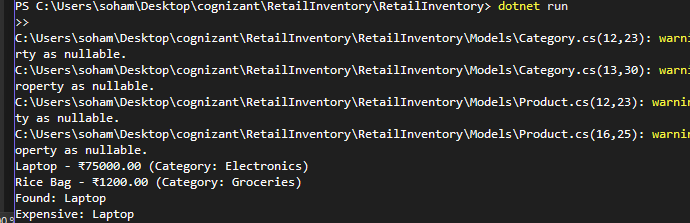
Console.WriteLine($"Found: {product?.Name}");

// 3. First Product with Price > ₹50,000

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

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

**OUTPUT:**

****