EF Core 8.0 Guided Hands-On Exercises

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

**What is ORM?**

**ORM (Object-Relational Mapping)** is a programming approach that connects C# classes with database tables, allowing developers to interact with databases using objects instead of raw SQL queries.

Every class in C# typically corresponds to a table in the database.

Properties within the class align with columns in that table.

Relationships such as one-to-many or many-to-many are handled using navigation properties between classes.

**Benefits of ORM:**

**Increased Productivity:** Developers can perform database operations using C# code directly, reducing the need to write SQL queries manually for common tasks.

**Easier Maintenance:** When the data model evolves, the corresponding changes can be automatically applied to the database using features like migrations.

**Higher Abstraction:** By working with C# objects instead of writing raw SQL, developers can focus more on business logic, which reduces the chance of errors and simplifies the codebase.

**EF Core vs EF Framework:**

**EF Core:**

* Works across platforms it supports Windows, Linux, and macOS with .NET 6/7/8.
* Designed to be lightweight and modular, and it supports modern capabilities like LINQ, asynchronous operations, and compiled queries.
* It’s under active development and is the preferred choice for building new applications.

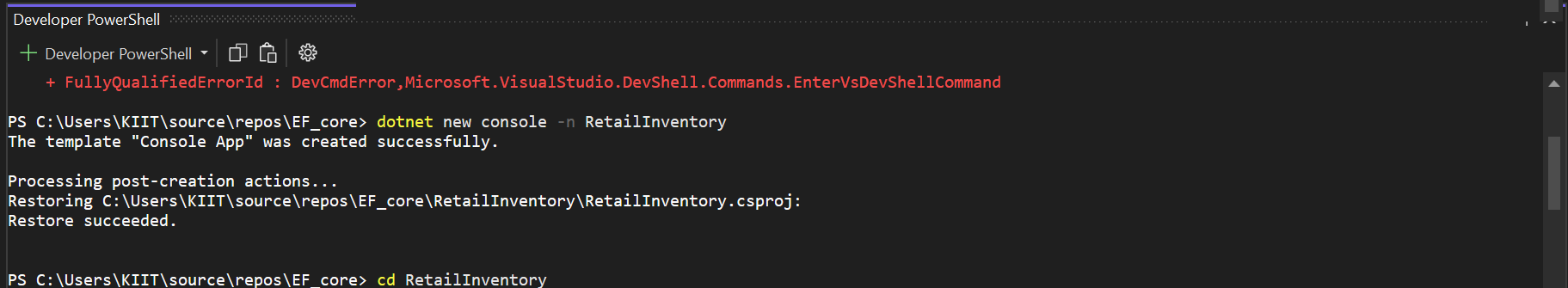
**Entity Framework 6 (EF6):**

* Limited to Windows-based applications as it only runs on the .NET Framework.
* Although it's more mature and stable, it doesn't offer the flexibility or portability of EF Core.
* Missing several modern features that are now considered standard, making it less ideal for new development.

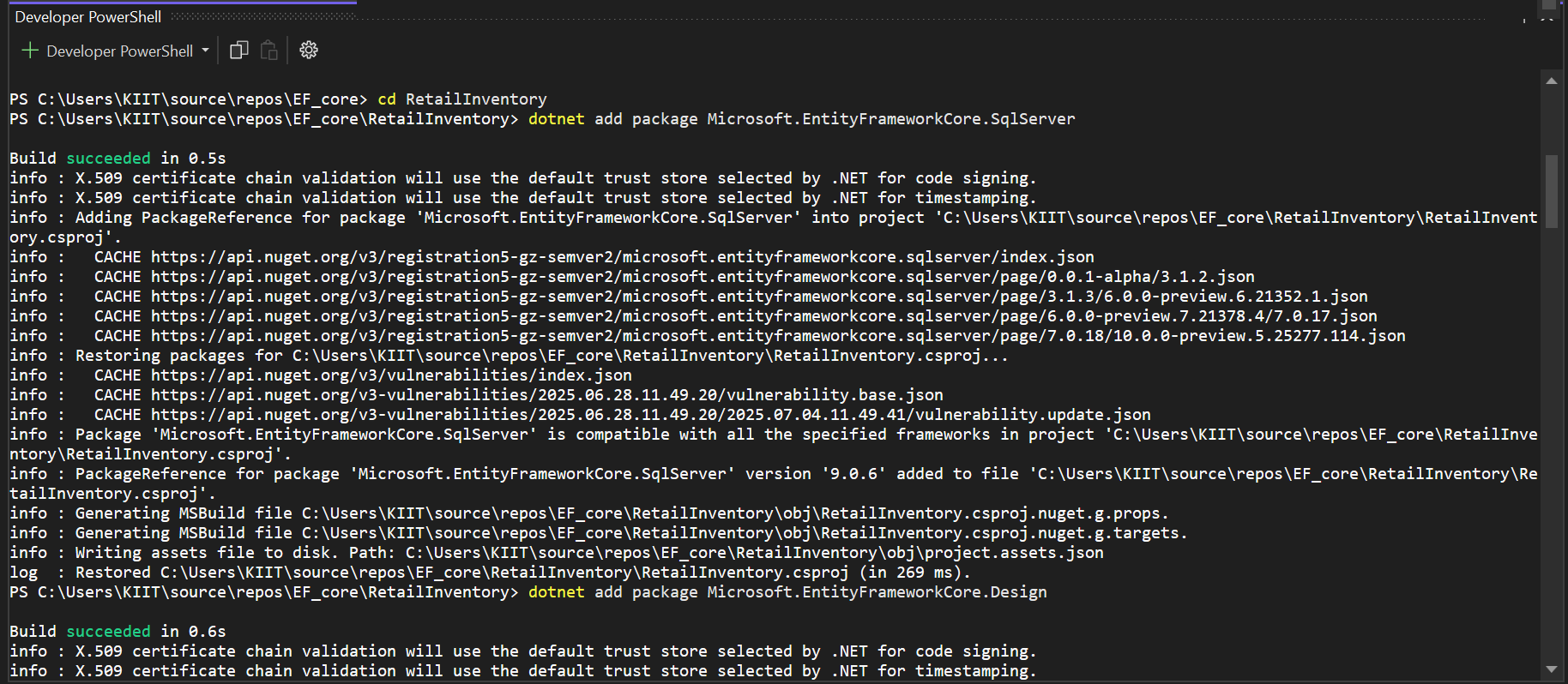
**EF Core 8.0 Features:**

* **JSON Column Support:** Enables mapping and querying of JSON data stored in SQL Server columns, allowing for more flexible and complex data structures.
* **Compiled Models:** Improves application startup time and query performance by precompiling the data model.
* **Interceptors:** Lets you tap into EF Core’s internal processes to implement custom behaviors like logging, validation, or auditing during database operations.
* **Bulk Operations:** Enhances performance for batch insert, update, and delete operations, making it easier to handle large datasets efficiently.

**Create a .NET Console App:**

****

**Install EF Core Packages:**

****

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

**Create Models:**

namespace Models;

public class Category

{

public int CategoryId { get; set; }

public required string CategoryName { get; set; }

public ICollection<Product> ProductList { get; set; } = new List<Product>();

}

public class Product

{

public int ProductId { get; set; }

public required string ProductName { get; set; }

public decimal UnitPrice { get; set; }

public int CategoryId { get; set; }

public required Category Category { get; set; }

}

**Create AppDbContext:**

using Microsoft.EntityFrameworkCore;

using Models;

public class AppDbContext : DbContext

{

public DbSet<Product> Products => Set<Product>();

public DbSet<Category> Categories => Set<Category>();

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseSqlServer(

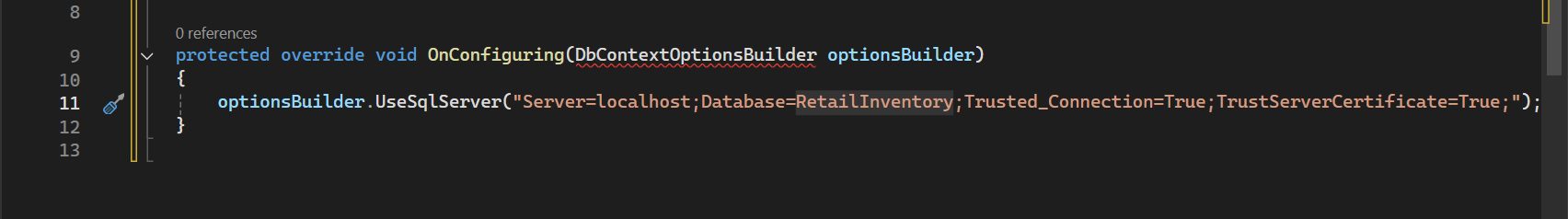
"Server=localhost;Database=RetailInventory;Trusted\_Connection=True;TrustServerCertificate=True;");

}

}

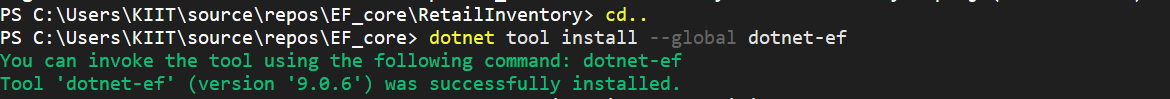
}

**Add Connection String**

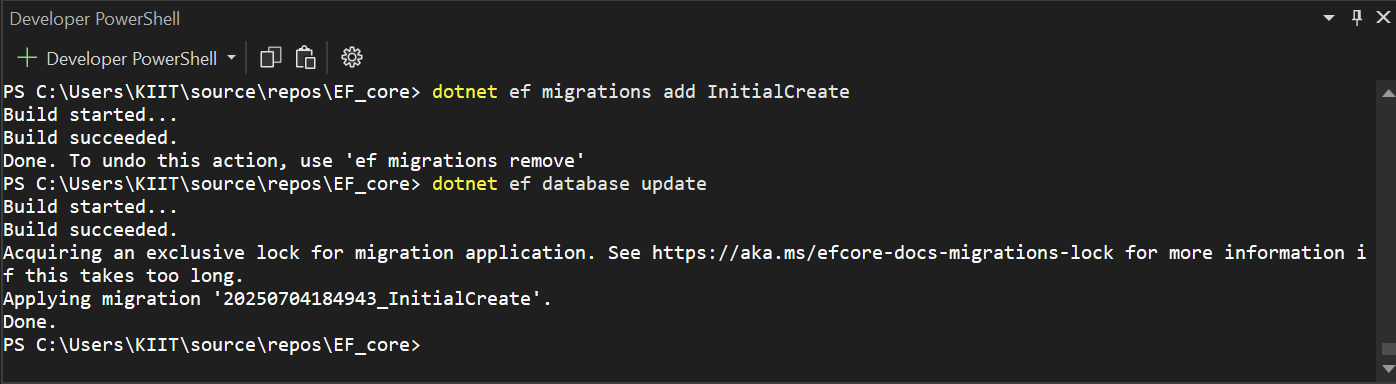


# Lab 3: Using EF Core CLI to Create and Apply Migrations

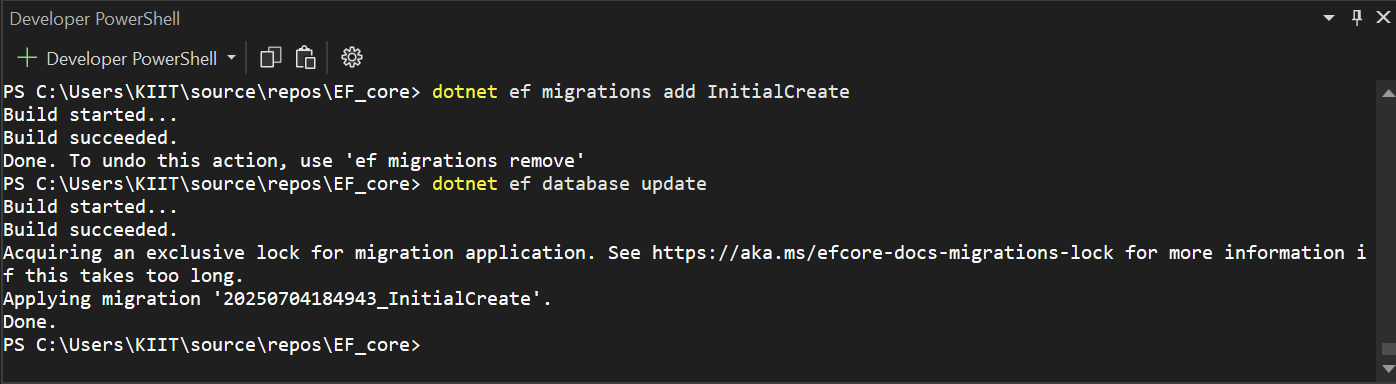
**Install EF Core CLI**

****

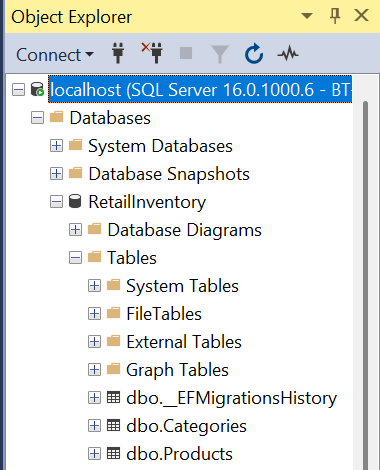
**Create Initial Migration**



**Apply Migration to Create Database:**



**Verify in SQL Server:**

****

# Lab 4: Inserting Initial Data into the Database

**Insert Data in Program.cs:**

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.EntityFrameworkCore;

using Models;

namespace RetailInventory;

public class Program

{

public static async Task Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Register services

builder.Services.AddDbContext<AppDbContext>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure middleware

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

// Seed initial data if necessary

using var scope = app.Services.CreateScope();

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

if (!dbContext.Categories.Any())

{

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

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

dbContext.Categories.AddRange(electronics, groceries);

await dbContext.SaveChangesAsync();

var products = new List<Product>

{

new() { Name = "Laptop", Price = 1200.00m, CategoryId = electronics.Id, Category = electronics },

new() { Name = "Apple", Price = 1.50m, CategoryId = groceries.Id, Category = groceries }

};

dbContext.Products.AddRange(products);

await dbContext.SaveChangesAsync();

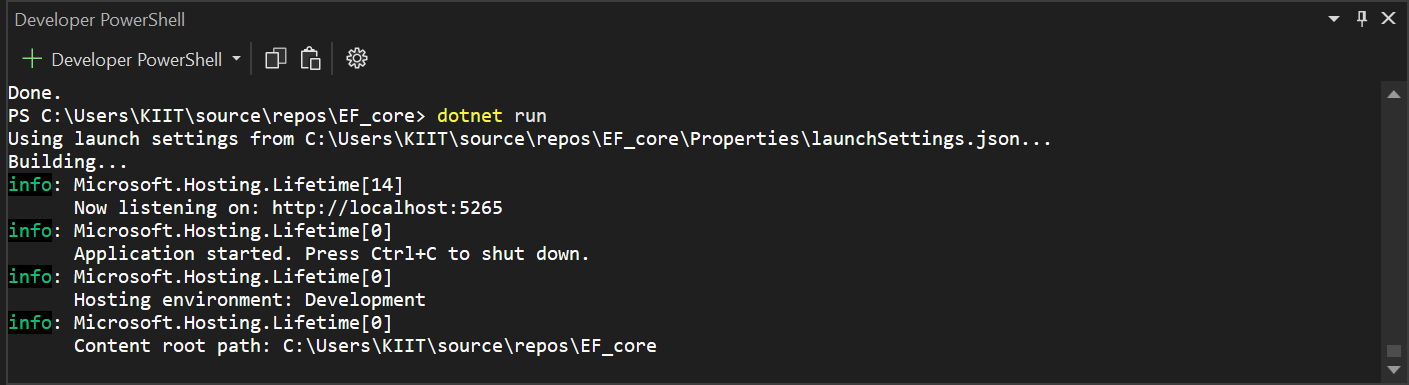
}

await app.RunAsync();

}

}

**Run the App:**

****

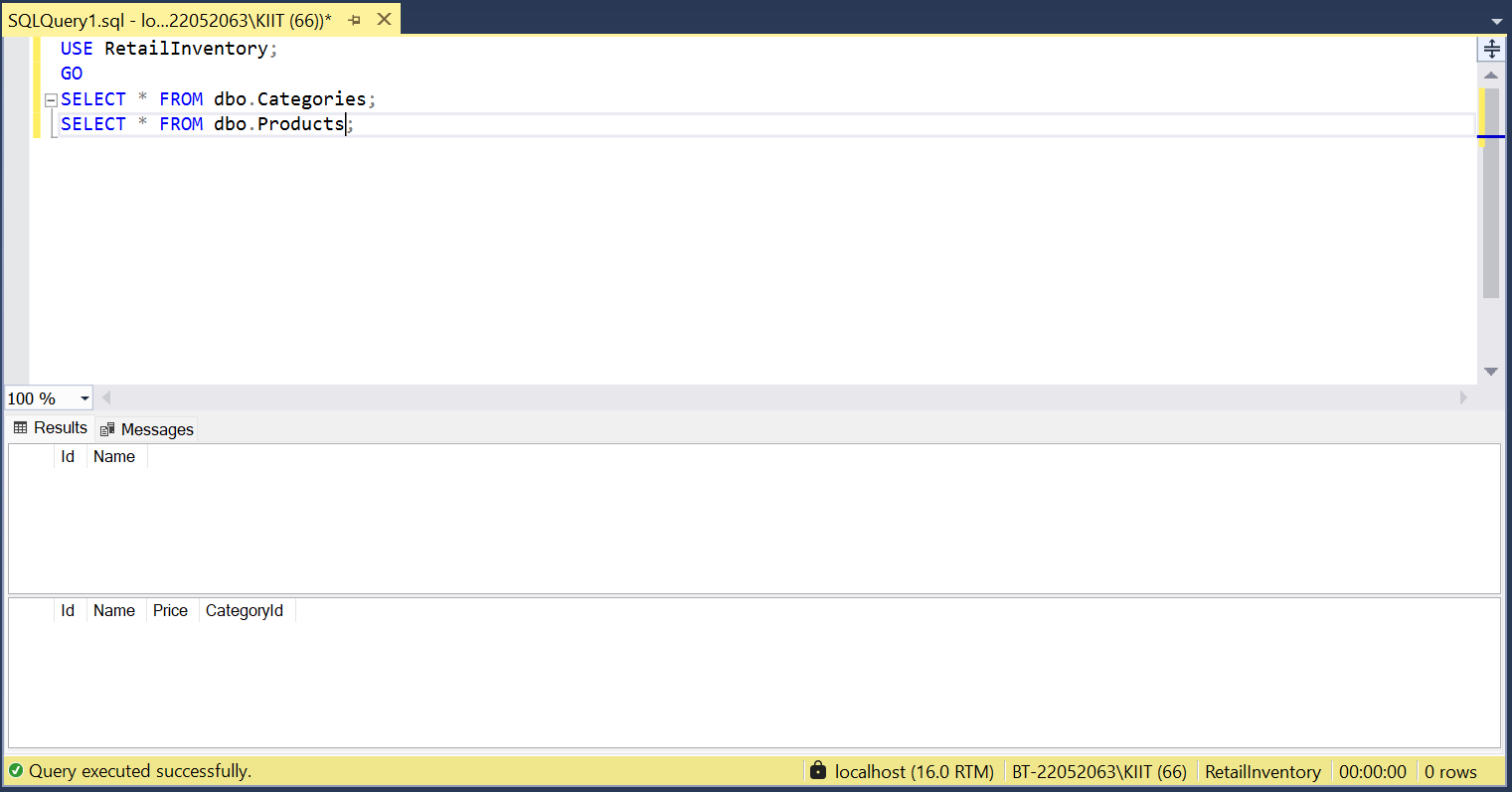
**Verify in SQL Server:**

USE RetailInventory;

GO

SELECT \* FROM dbo.Categories;

SELECT \* FROM dbo.Products;

****

# Lab 5: Retrieving Data from the Database

**Retrieve All Products, Find by ID, FirstOrDefault with Condition:**

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.EntityFrameworkCore;

using Models;

namespace RetailInventory;

public class Program

{

public static async Task Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Register services

builder.Services.AddDbContext<AppDbContext>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure middleware

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

// Seed initial data if not present

using (var scope = app.Services.CreateScope())

{

var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();

if (!db.Categories.Any())

{

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

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

db.Categories.AddRange(electronics, groceries);

await db.SaveChangesAsync();

var products = new List<Product>

{

new() { Name = "Laptop", Price = 1200.00m, CategoryId = electronics.Id, Category = electronics },

new() { Name = "Apple", Price = 1.50m, CategoryId = groceries.Id, Category = groceries }

};

db.Products.AddRange(products);

await db.SaveChangesAsync();

}

}

// Retrieve and display data to console

using (var scope = app.Services.CreateScope())

{

var context = scope.ServiceProvider.GetRequiredService<AppDbContext>();

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

Console.WriteLine("Product List:");

foreach (var p in products)

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

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

Console.WriteLine($"\nFound Product with ID 1: {product?.Name ?? "Not found"}");

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

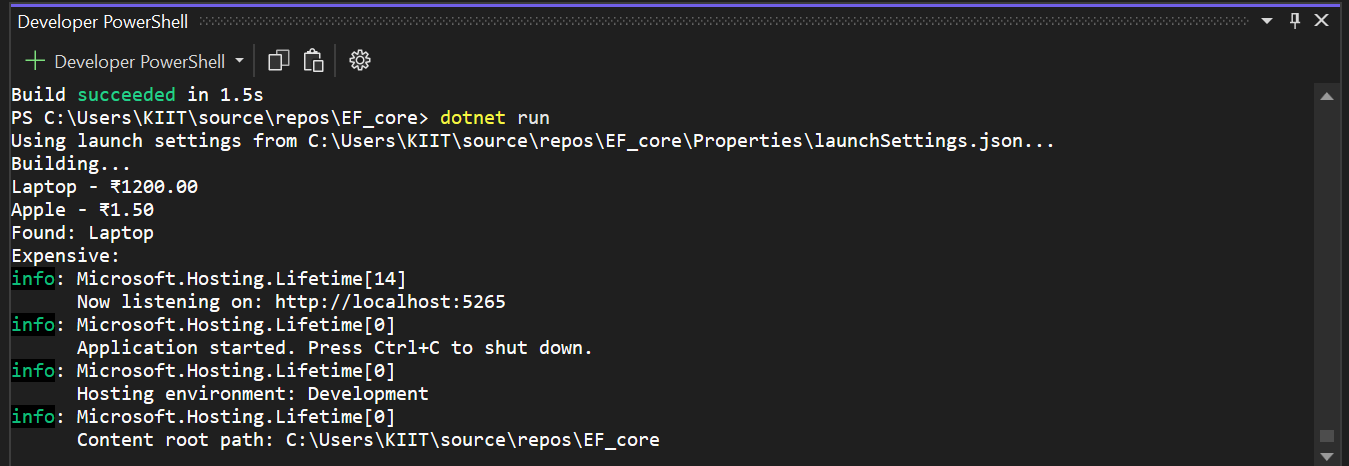
Console.WriteLine($"First expensive product (> ₹50000): {expensive?.Name ?? "None found"}");

}

await app.RunAsync();

}

}

****