Access Database in ASP.NET Blazor

What is EF Core?

- Entity Framework Core is Microsoft’s modern object-relational mapper (ORM) for .NET.
- It lets you interact with a relational database using C# classes instead of SQL.
- In Blazor, EF Core is used on the server-side and is injected into components for CRUD operations.

Install Required EF Core Packages

- Install the following two EF Core packages:
  - Microsoft.EntityFrameworkCore.SqlServer
  - Microsoft.EntityFrameworkCore.Tools

Define the Model Class

- Create a folder named Models and add a file named Product.cs:

```
public class Product
{
    public int Id { get; set; }  // Primary Key
    public string Name { get; set; }
    public double Price { get; set; }
}
```

Create DbContext

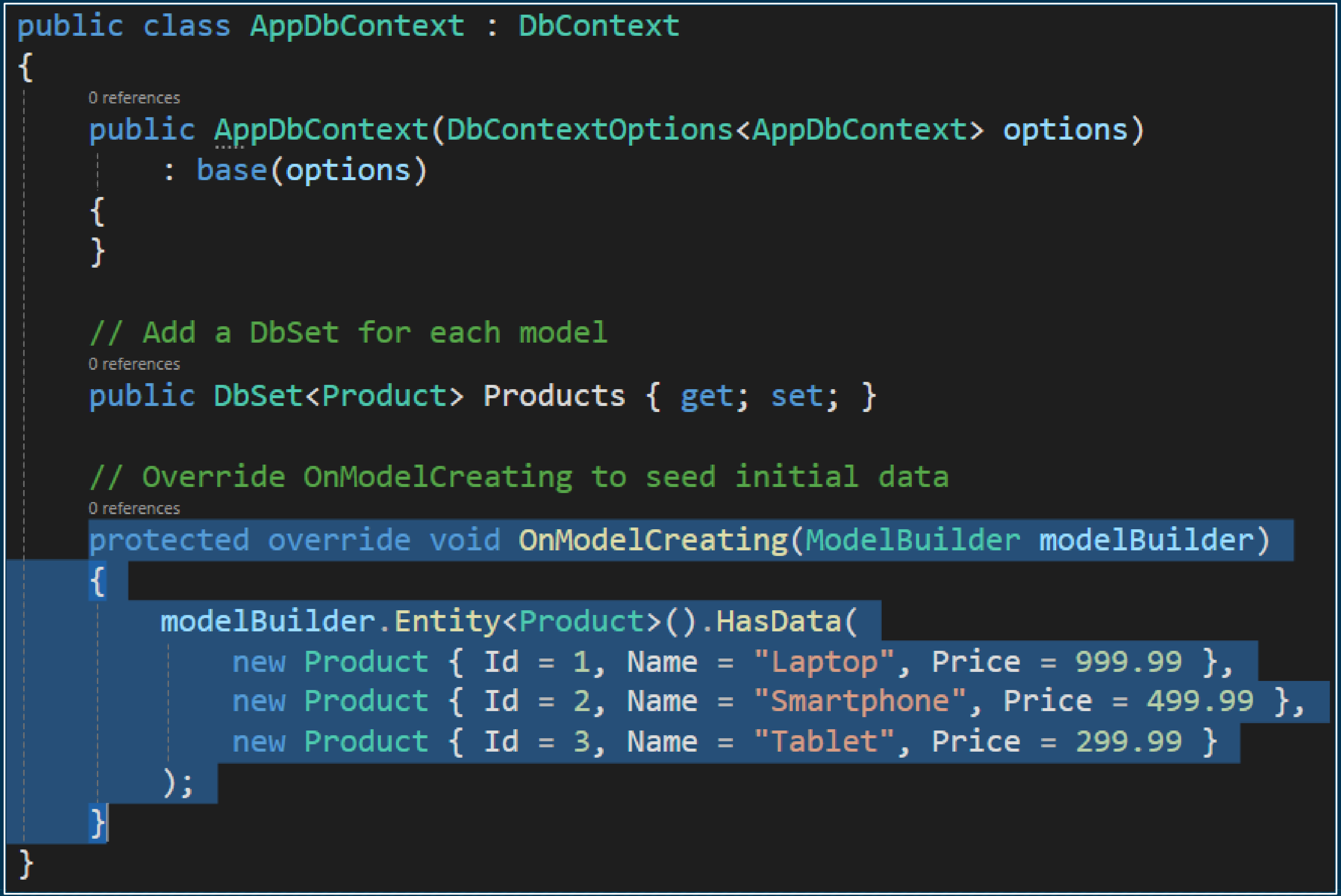
- Now create a folder called Data, and inside it, add a class called AppDbContext.cs:

```
using Microsoft.EntityFrameworkCore;
using YourProjectName.Models;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }

    // Add a DbSet for each model
    public DbSet<Product> Products { get; set; }
}
```

- `DbContextOptions<T>` tells EF how to connect/configure the database.
- `DbSet<Product>` represents the table of products in the database.

Data Seed the Products Table

- Override the `OnModelCreating` method to data seed the Products table.



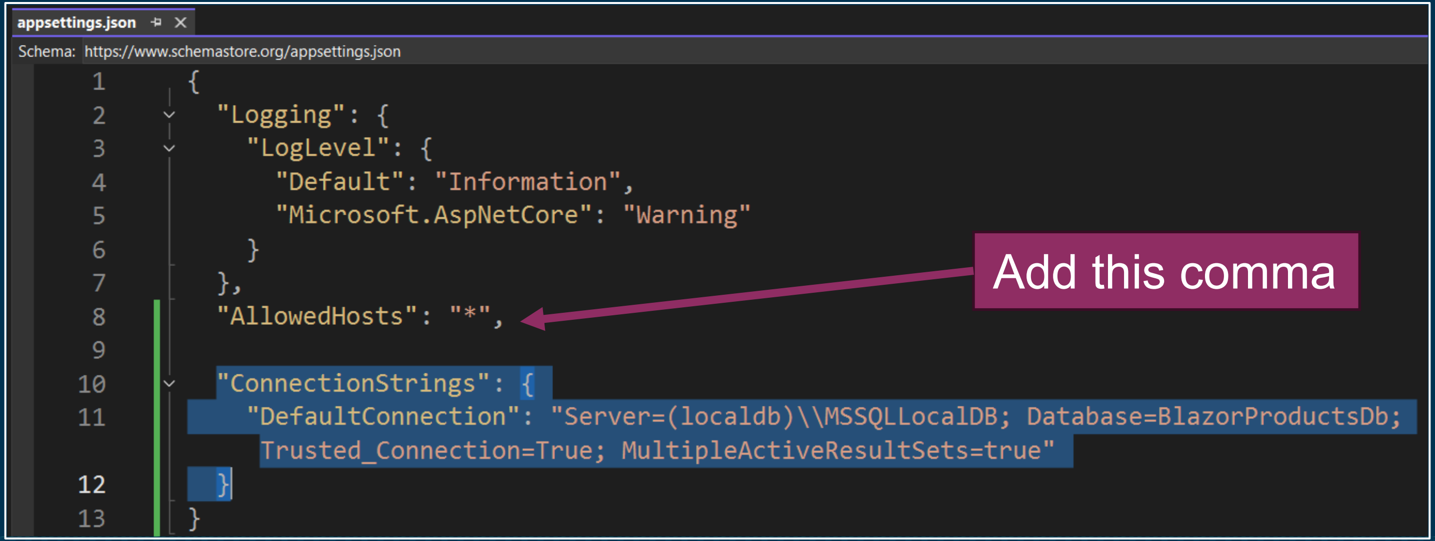
```
public class AppDbContext : DbContext
{
    0 references
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }

    // Add a DbSet for each model
    0 references
    public DbSet<Product> Products { get; set; }

    // Override OnModelCreating to seed initial data
    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().HasData(
            new Product { Id = 1, Name = "Laptop", Price = 999.99 },
            new Product { Id = 2, Name = "Smartphone", Price = 499.99 },
            new Product { Id = 3, Name = "Tablet", Price = 299.99 }
        );
    }
}
```

Sheridan Get Creative

Add the Connection String

- In appsettings.json, add:

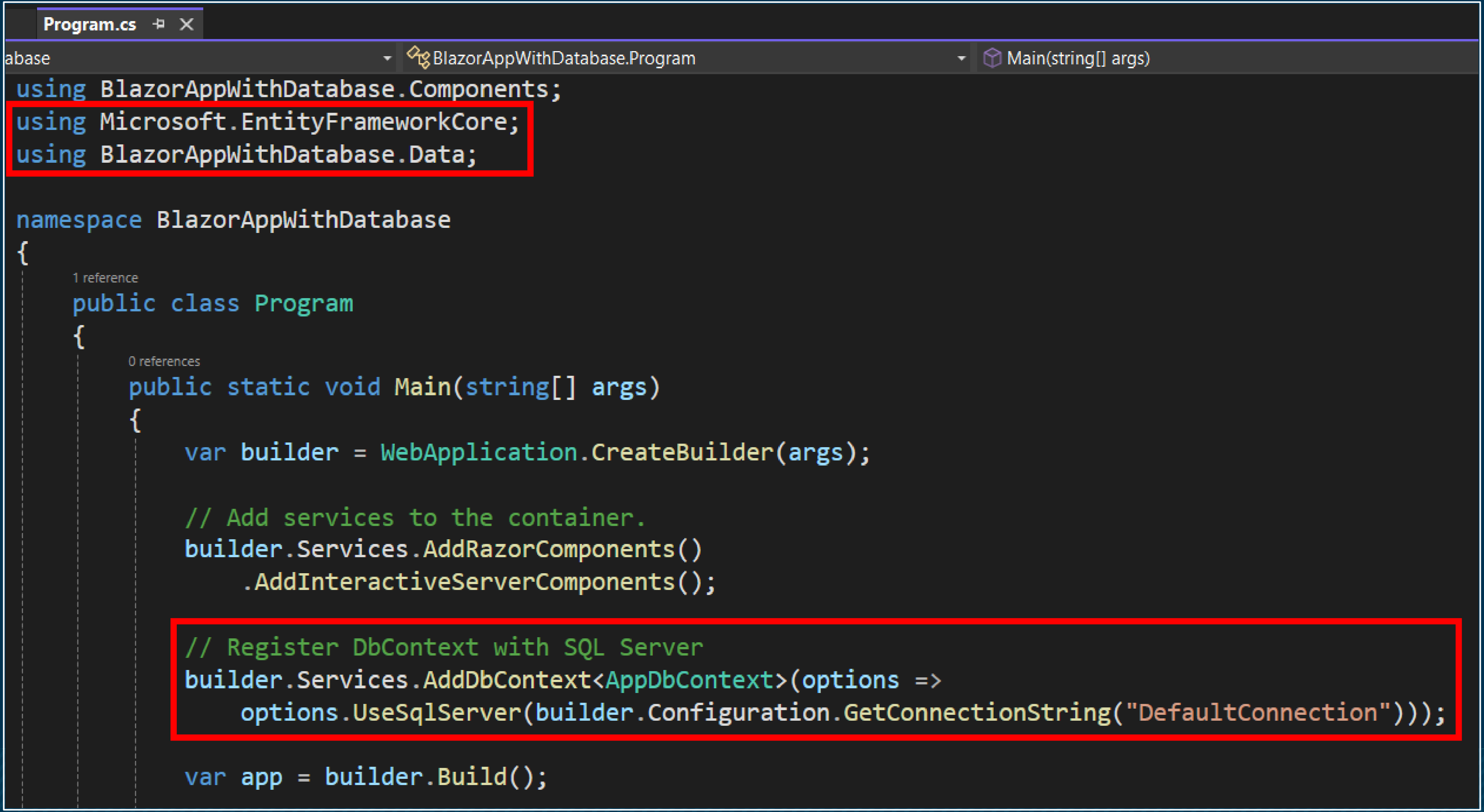
```
"ConnectionStrings": {
  "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;
                        Database=BlazorProductsDb;
                        Trusted_Connection=True;
                        MultipleActiveResultSets=true"
}
```



```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",

  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB; Database=BlazorProductsDb;
      Trusted_Connection=True; MultipleActiveResultSets=true"
  }
}
```

Register DbContext in DI (Dependency Injection)

- Open Program.cs and register `AppDbContext` in the DI container, before the `builder.Build()`. Also, include the required namespaces.

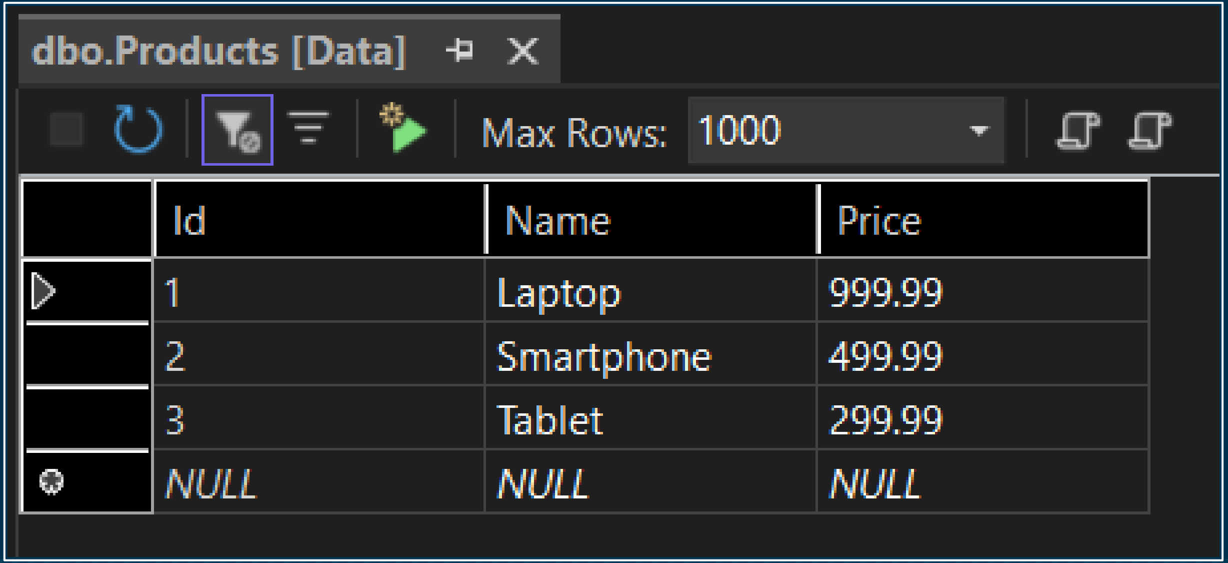


```
using BlazorAppWithDatabase.Components;
using Microsoft.EntityFrameworkCore;
using BlazorAppWithDatabase.Data;

namespace BlazorAppWithDatabase
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);

            // Add services to the container.
            builder.Services.AddRazorComponents()
                .AddInteractiveServerComponents();

            // Register DbContext with SQL Server
            builder.Services.AddDbContext<AppDbContext>(options =>
                options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

            var app = builder.Build();
```

Create & Apply Migrations

- Now, run the EF Core migration commands to generate the database schema from your model:

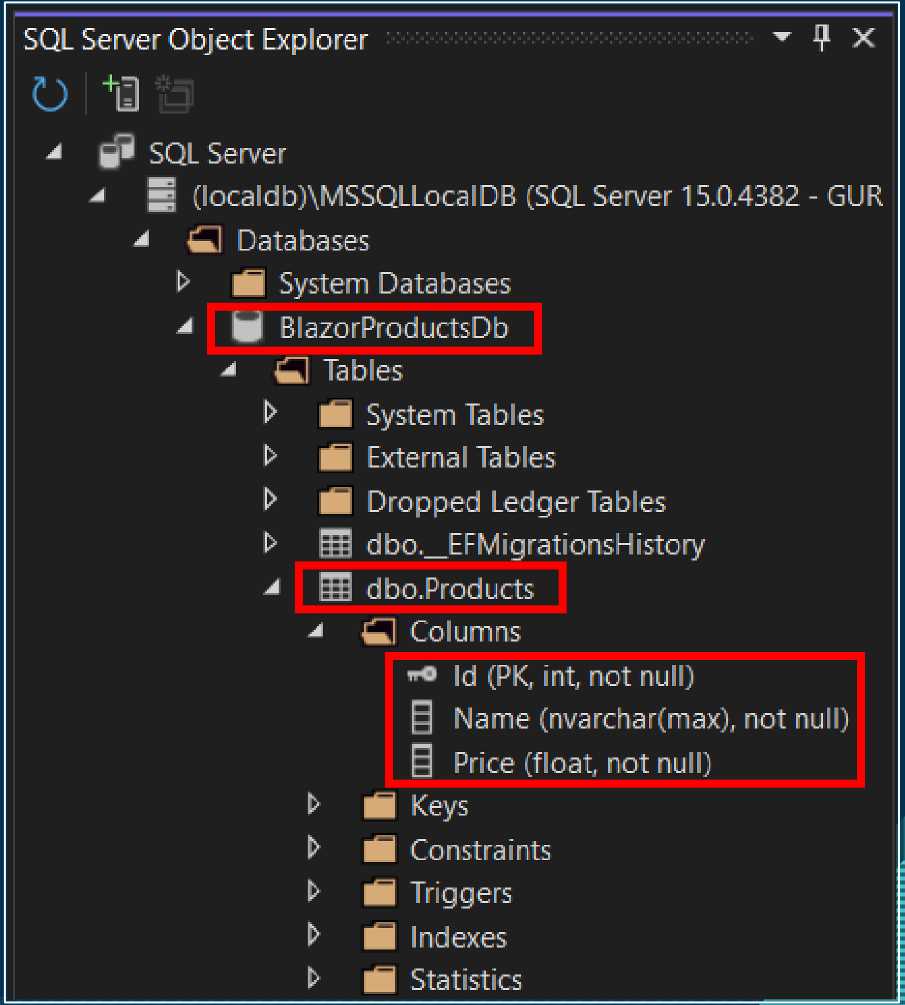
```
PM> add-migration InitialMigration

PM> update-database
```



dbo.Products [Data]

Max Rows: 1000

Id	Name	Price
1	Laptop	999.99
2	Smartphone	499.99
3	Tablet	299.99
NULL	*NULL*	*NULL*

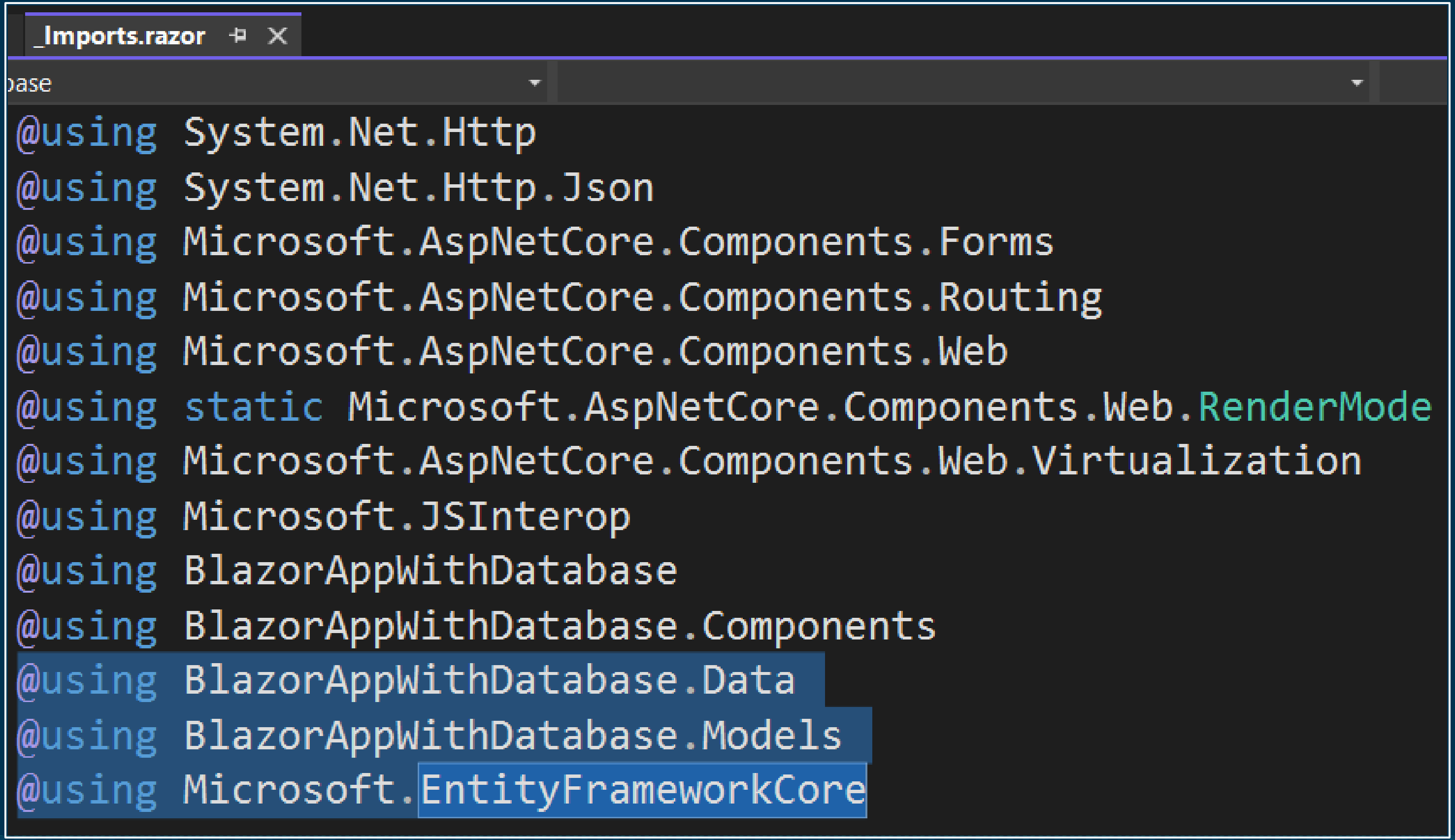


SQL Server Object Explorer

```
SQL Server
  (localdb)\MSSQLLocalDB (SQL Server 15.0.4382 - GUR
    Databases
      System Databases
      BlazorProductsDb
        Tables
          System Tables
          External Tables
          Dropped Ledger Tables
          dbo.__EFMigrationsHistory
          dbo.Products
            Columns
              Id (PK, int, not null)
              Name (nvarchar(max), not null)
              Price (float, not null)
            Keys
            Constraints
            Triggers
            Indexes
            Statistics
```

Include the Namespaces in the _Imports.razor

- Open Components/_Imports.razor file.
- Include the highlighted namespaces.
- Switch the `BlazorAppWithDatabase` with your project name.

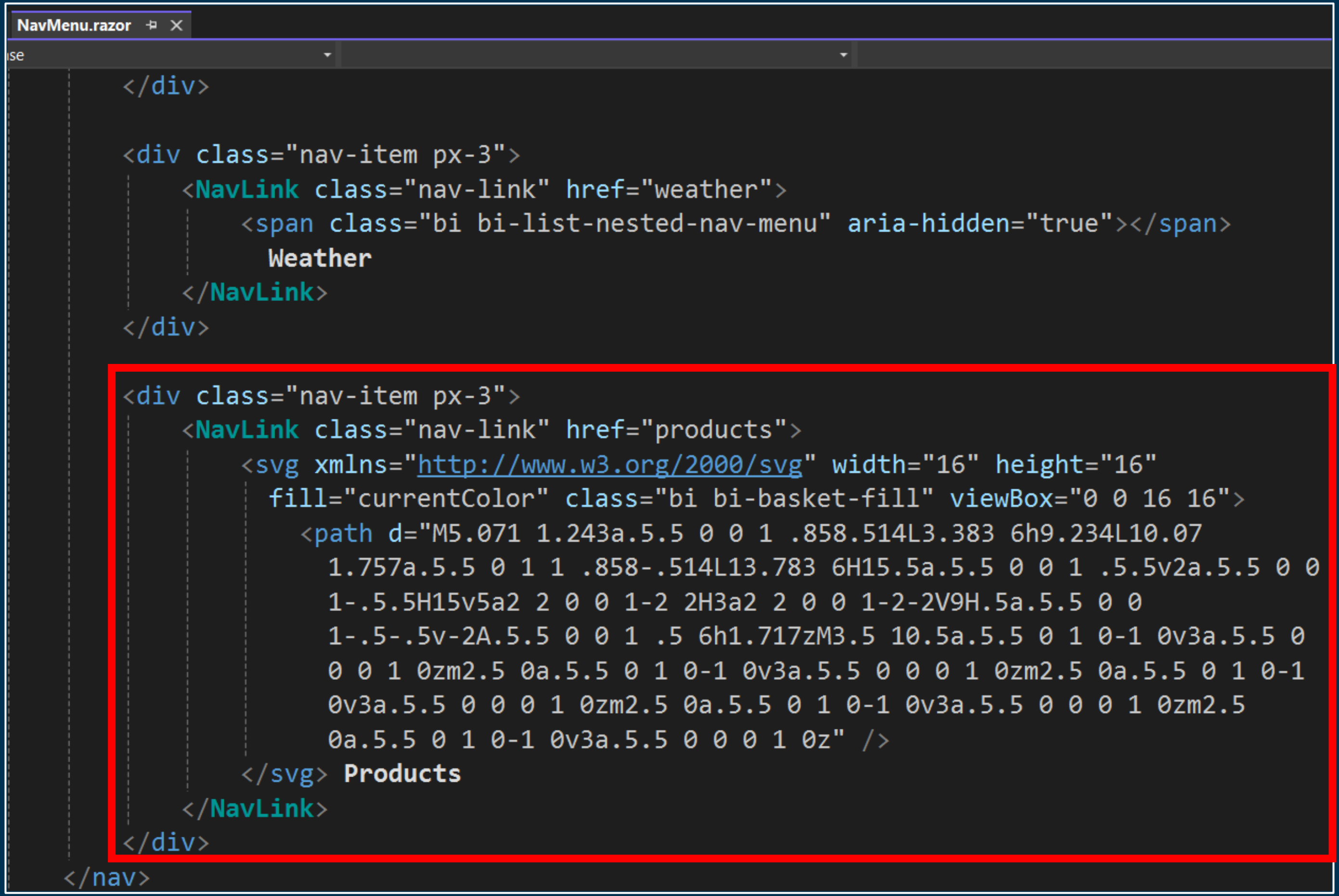


_Imports.razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using static Microsoft.AspNetCore.Components.Web.RenderMode
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using BlazorAppWithDatabase
@using BlazorAppWithDatabase.Components
@using BlazorAppWithDatabase.Data
@using BlazorAppWithDatabase.Models
@using Microsoft.EntityFrameworkCore
```

Include the Namespaces in the _Imports.razor

- Add navigation link to the Components/Layout/NavMenu.razor:



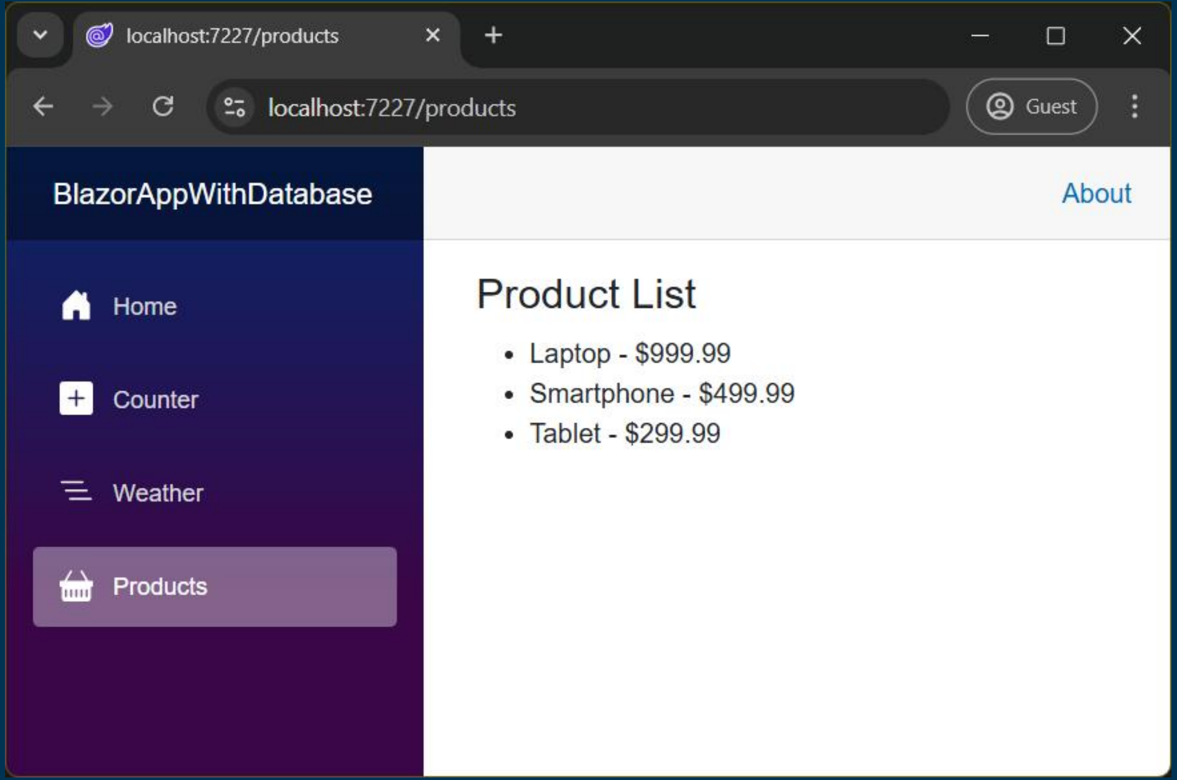
```
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="weather">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
                 Weather
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="products">
                <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16"
                  fill="currentColor" class="bi bi-basket-fill" viewBox="0 0 16 16">
                    <path d="M5.071 1.243a.5.5 0 0 1 .858.514L3.383 6h9.234L10.07
                      1.757a.5.5 0 1 1 .858-.514L13.783 6H15.5a.5.5 0 0 1 .5.5v2a.5.5 0 0
                      1-.5.5H15v5a2 2 0 0 1-2 2H3a2 2 0 0 1-2-2V9H.5a.5.5 0 0
                      1-.5-.5v-2A.5.5 0 0 1 .5 6h1.717zM3.5 10.5a.5.5 0 1 0-1 0v3a.5.5 0
                      0 0 1 0zm2.5 0a.5.5 0 1 0-1 0v3a.5.5 0 0 0 1 0zm2.5 0a.5.5 0 1 0-1
                      0v3a.5.5 0 0 0 1 0zm2.5 0a.5.5 0 1 0-1 0v3a.5.5 0 0 0 1 0zm2.5
                      0a.5.5 0 1 0-1 0v3a.5.5 0 0 0 1 0z" />
                </svg> Products
            </NavLink>
        </div>
    </nav>
```

- I'm using this icon: https://icons.getbootstrap.com/icons/basket-fill/

Create a Razor Component

- Under Components → Pages, add a Razor Component, ProductList.razor:

```
@page "/products"
@inject AppDbContext Db

<h3>Product List</h3>

@if (products == null) {
    <p>No products found</p>
}
else {
    <ul>
        @foreach (var p in products)
        {
            <li>@p.Name - @p.Price.ToString("C")</li>
        }
    </ul>
}

@code {
    private List<Product>? products;
    protected override void OnInitialized()
    {
        products = Db.Products.ToList();
    }
}
```



Product List

- Laptop - $999.99
- Smartphone - $499.99
- Tablet - $299.99

Explanation of ProductList.razor

- `@page "/products"`:
  - This makes the Razor component accessible via the URL `/products`.
- `@inject AppDbContext Db`:
  - This injects the `AppDbContext` into the component.
  - `Db` is the proprety name you'll use to access the database.
  - Without this line, you wouldn't be able to call `Db.Products`.
- `@foreach` loop:
  - Uses Razor's `@foreach` loop to render each product.

```
@page "/products"
@inject AppDbContext Db

<h3>Product List</h3>

@if (products == null) {
    <p>No products found</p>
}
else {
    <ul>
        @foreach (var p in products)
        {
            <li>@p.Name - $@p.Price</li>
        }
    </ul>
}

@code {
    private List<Product>? products;
    protected override void OnInitialized()
    {
        products = Db.Products.ToList();
    }
}
```

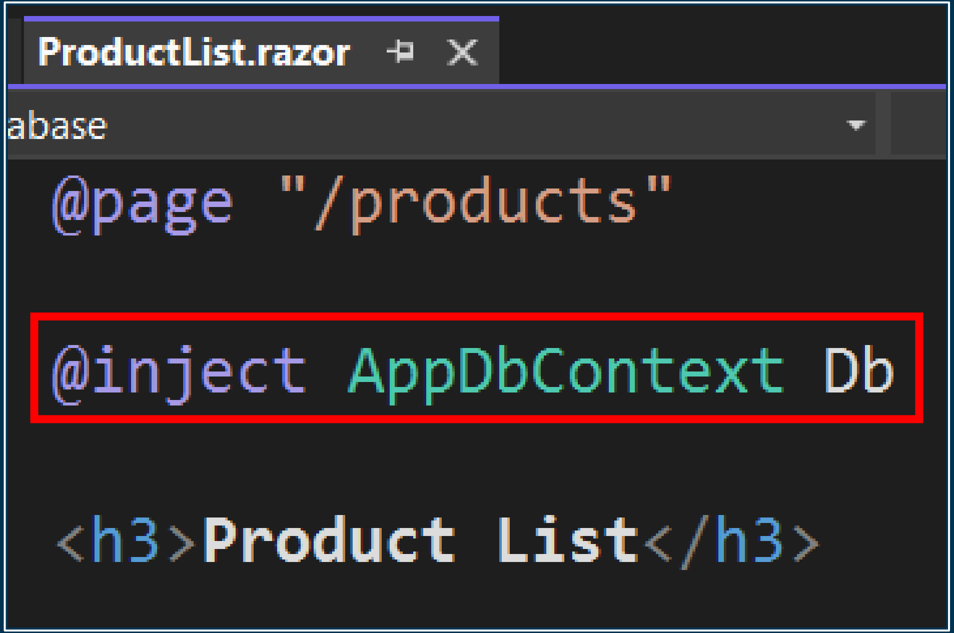
Explanation of ProductList.razor

- `@code { ... }`:
  - This is the C# logic block of the component.
- `List<Product> products`:
  - This variable stores the list of products fetched from the database.
- `OnInitialized()` method:
  - This is a Blazor lifecycle method.
  - It runs when the component is first initialized.
- `products = Db.Products.ToList()`:
  - EF Core call to retrieve all rows from the Products table.

```
@page "/products"
@inject AppDbContext Db

<h3>Product List</h3>

@if (products == null) {
    <p>No products found</p>
}
else {
    <ul>
        @foreach (var p in products)
        {
            <li>@p.Name - $@p.Price</li>
        }
    </ul>
}

@code {
    private List<Product>? products;
    protected override void OnInitialized()
    {
        products = Db.Products.ToList();
    }
}
```

What is @inject?

- `@inject` is a Blazor directive used to request a service instance from the dependency injection (DI) container.
- It tells Blazor to inject an instance of `AppDbContext` (your database context) into the component.
- `Db` becomes the property you can use to call EF Core methods like `Db.Products.ToList()`.



```
@page "/products"

@inject AppDbContext Db

<h3>Product List</h3>
```

- Why `Db` and not `db`?
  - Because when you use `@inject`, it injects the value as a public property.
  - And by C# convention, public properties are PascalCase.
  - Even though `Db` looks like a variable, it's a property behind the scenes:

```
[Inject]
public AppDbContext Db { get; set; }
```

Why is @inject needed in Blazor?

- It is because Blazor components are not like typical C# classes that accept constructor parameters.
- Instead:
  - Blazor uses property injection, not constructor injection.
  - So, we use `@inject` to tell the framework:
    - “Please give me an instance of this service.”
- This applies to services like:
  - `AppDbContext` (for EF Core)
  - `HttpClient` (for web APIs)
  - Custom services (like `ProductService`, `AuthService`, etc.)
- Is it required to use `@inject`?
  - Yes, if you're using services inside a .razor file, you need `@inject` to access them.

Dependency Injection (DI)

- **Dependency Injection (DI)** is a technique where dependencies (e.g., services, repositories) are injected into a class rather than being created inside the class.
- This promotes inversion of control (IoC), meaning that the class does not instantiate its dependencies but instead receives them from an external source.

- **Why Use DI?**
  - Reduces tight coupling between components.
  - Improves testability (easier to mock dependencies in unit tests).
  - Enhances maintainability (easier to change dependencies without modifying dependent classes).

Difference Between Using DI and Not Using DI

- Let's say we have a `Car` class that depends on an `Engine` class.
- We'll compare two approaches:
  - Without Dependency Injection (Tightly Coupled)
  - With Dependency Injection (Loosely Coupled & Maintainable)

Without Dependency Injection (Tightly Coupled)

- Here, the `Car` class creates an instance of `Engine` inside itself.

```csharp
// Engine class
public class Engine
{
    public string Start()
    {
        return "Engine started!";
    }
}
```

```csharp
// Car class tightly coupled to Engine
public class Car
{
    private Engine _engine;

    public Car()
    {
        _engine = new Engine(); // Direct dependency (Bad Practice)
    }

    public void Drive()
    {
        Console.WriteLine(_engine.Start());
        Console.WriteLine("Car is moving...");
    }
}
```

```csharp
// Main Program
public class Program
{
    public static void Main()
    {
        Car car = new Car();  // Creates its own dependencies
        car.Drive();
    }
}
```

Problems with this Approach

- Tightly coupled:
  - The `Car` class is directly dependent on `Engine`.
- Difficult to replace:
  - If we want to change the `Engine` (e.g., `ElectricEngine`), we have to modify the `Car` class.
- Not testable:
  - We cannot replace `Engine` with a mock in unit testing.

With Dependency Injection (Loosely Coupled)

- Here, the `Car` class does not create an `Engine` instance but instead receives it as a dependency.

```csharp
// Engine interface
public interface IEngine
{
    string Start();
}

// Petrol Engine implementation
public class PetrolEngine : IEngine
{
    public string Start()
    {
        return "Petrol Engine started!";
    }
}
```

```csharp
// Car class with Dependency Injection
public class Car
{
    private IEngine _engine;

    // Engine is injected via constructor
    public Car(IEngine engine)
    {
        _engine = engine;
    }

    public void Drive()
    {
        Console.WriteLine(_engine.Start());
        Console.WriteLine("Car is moving...");
    }
}
```

```csharp
// Main Program with DI
public class Program
{
    public static void Main()
    {
        IEngine petrolEngine = new PetrolEngine(); // We decide which engine to use
        Car car = new Car(petrolEngine);           // Inject dependency
        car.Drive();
    }
}
```

Sheridan Get Creative

Benefits of Dependency Injection

- Loosely Coupled:
  - `Car` depends on `IEngine`, not a specific `Engine` implementation.
  - We can easily replace `PetrolEngine` with another engine type.
- Easier to Modify:
  - If we want to add an `ElectricEngine`, no need to change the `Car` class.

```
public class ElectricEngine : IEngine
{
    public string Start()
    {
        return "Electric Engine started!";
    }
}
```

```
// Main Program with DI
public class Program
{
    public static void Main()
    {
        IEngine petrolEngine = new PetrolEngine();
        Car car = new Car(petrolEngine);
        car.Drive();

        IEngine electricEngine = new ElectricEngine();
        Car car = new Car(electricEngine);
        car.Drive();
    }
}
```

Sheridan Get Creative

Summary

Approach	Without DI (Tightly Coupled)	With DI (Loosely Coupled)
Dependency Management	Class creates its own dependencies	Dependencies are passed in
Flexibility	Hard to replace components	Easily replaceable components
Testing	Hard to mock dependencies	Easily mockable for unit tests
Maintainability	Code changes require modifying multiple classes	Code is modular and easy to maintain

- Using Dependency Injection makes applications more flexible, testable, and maintainable.

Introducing Service Class

- A service class in Blazor is a class that encapsulates reusable logic, such as:
  - Fetching data from a database or API.
  - Performing calculations.
  - Business logic.
  - Handling authentication, file uploads, etc.

- In the context of EF Core and Blazor, a data service class helps separate the database logic from the UI (Razor components).

Why Use a Service Class?

- If you're using EF Core like this directly in your Razor component:

```
@inject AppDbContext Db
```

- That works for small projects, but it can cause problems:

Problem	Explanation
Code duplication	Every component using Db.Products.ToList() writes the same logic
Hard to test	You can't easily mock or swap Db in unit tests
Hard to maintain	Changing the DB structure requires updates in many files
Tightly coupled	UI depends directly on the database layer

- A service class abstracts that logic into one place.
- Makes it clean, testable, and reusable.

Create the Service Class

- Create a folder named Services and add a file named ProductService.cs:

```
using Microsoft.EntityFrameworkCore;
using YourProjectName.Models;
using YourProjectName.Data;

public class ProductService
{
    private readonly AppDbContext _context;

    public ProductService(AppDbContext context)
    {
        _context = context;
    }

    public async Task<List<Product>> GetProductsAsync()
    {
        return await _context.Products.ToListAsync();
    }
}
```

Why Is GetProductsAsync() Asynchronous?

- Synchronous (bad for performance):
  - If you use `ToList()` in a Blazor Server app:
    - It blocks the thread while the query runs.
    - That thread can't serve any other requests.
    - In Blazor Server, that thread is part of a limited pool; blocking it means fewer users can use the app simultaneously.

- Asynchronous (preferred):
  - With `await`, the Blazor component gives up the thread until the query completes.
  - The thread is returned to the thread pool to serve other users.
  - When the database returns the result, the thread resumes processing.
  - This improves scalability and responsiveness, especially in real-time and concurrent environments like: Blazor Server, ASP.NET Core MVC APIs, SignalR.

await and async

- ToListAsync() method is asynchronous, so use the await operator to capture the returned data of this method.

```
public async Task<List<Product>> GetProductsAsync()
{
    return await _context.Products.ToListAsync();
}
```

- Because the ToListAsync() method uses await, use async in the GetProductsAsync() method's definition.
- In C#, any method marked async must return a Task, Task<T>, or ValueTask<T>, not a direct value like List<Product>.
- Now that GetProductsAsync() method is asynchronous, it is a good practice to suffix the name with Async.

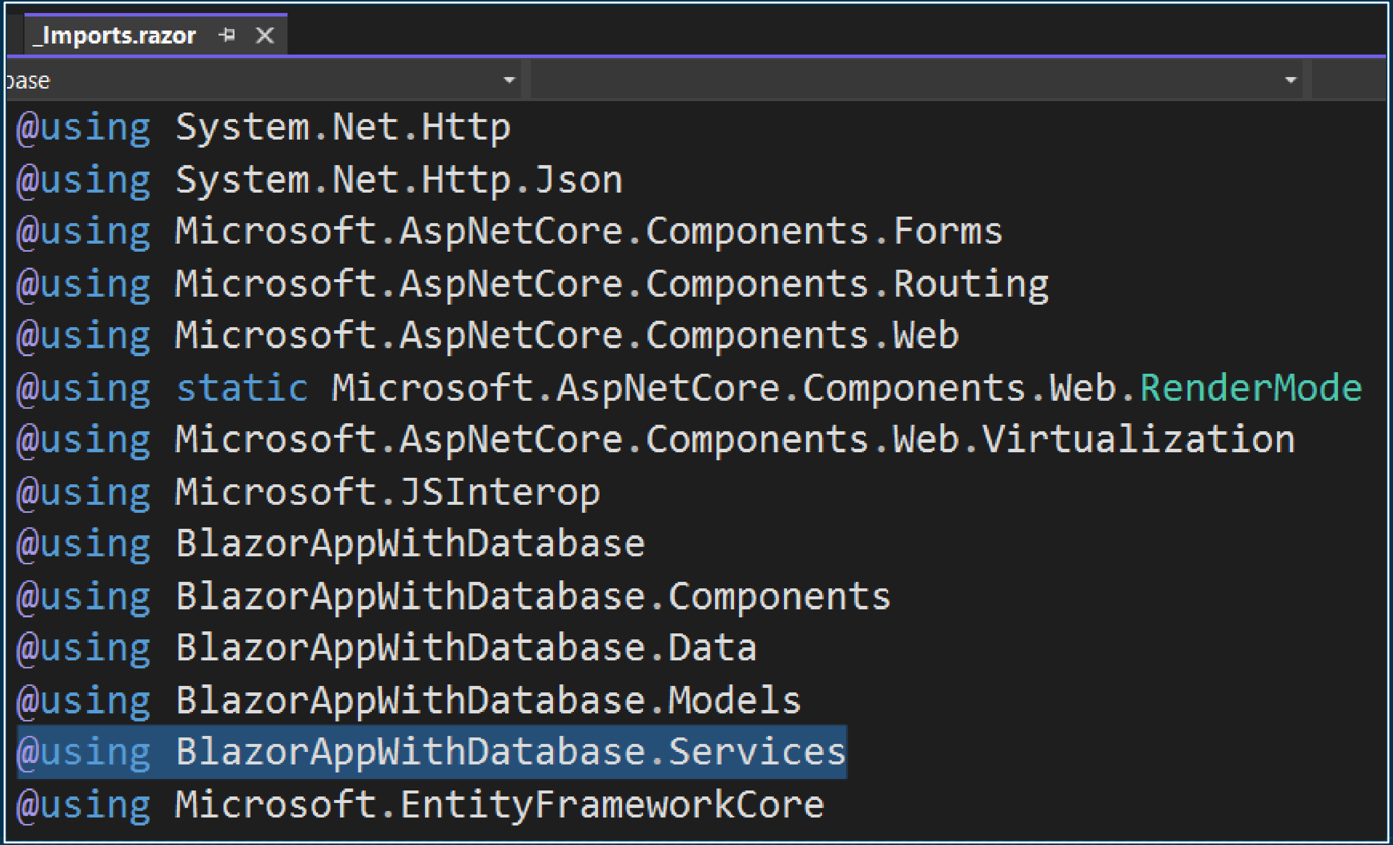
Register the Service in Program.cs

- Open Program.cs and register `ProductService`, before the `builder.Build()`.
- Also, include the required namespace.

```
builder.Services.AddScoped<ProductService>();
```

Include the Namespaces in the _Imports.razor

- Open Components/_Imports.razor file.
- Include the highlighted namespace.
- Switch the `BlazorAppWithDatabase` with your project name.



_Imports.razor

pase

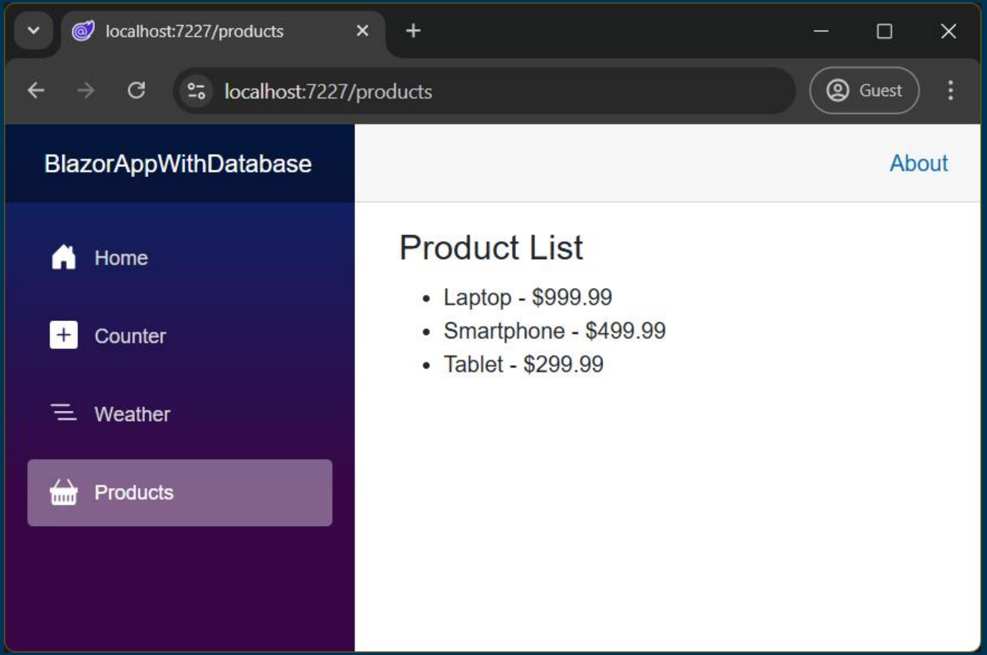
```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using static Microsoft.AspNetCore.Components.Web.RenderMode
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using BlazorAppWithDatabase
@using BlazorAppWithDatabase.Components
@using BlazorAppWithDatabase.Data
@using BlazorAppWithDatabase.Models
@using BlazorAppWithDatabase.Services
@using Microsoft.EntityFrameworkCore
```

Sheridan Get Creative

Use it in the Razor Component

- Go to ProductList.razor and modify the highlighted code.

```
@page "/products"
@inject ProductService ProductService

<h3>Product List</h3>

@if (products == null) {
    <p>No products found</p>
}
else {
    <ul>
        @foreach (var p in products)
        {
            <li>@p.Name - @p.Price.ToString("C")</li>
        }
    </ul>
}

@code {
    private List<Product>? products;
    protected override async Task OnInitializedAsync()
    {
        products = await ProductService.GetProductsAsync();
    }
}
```

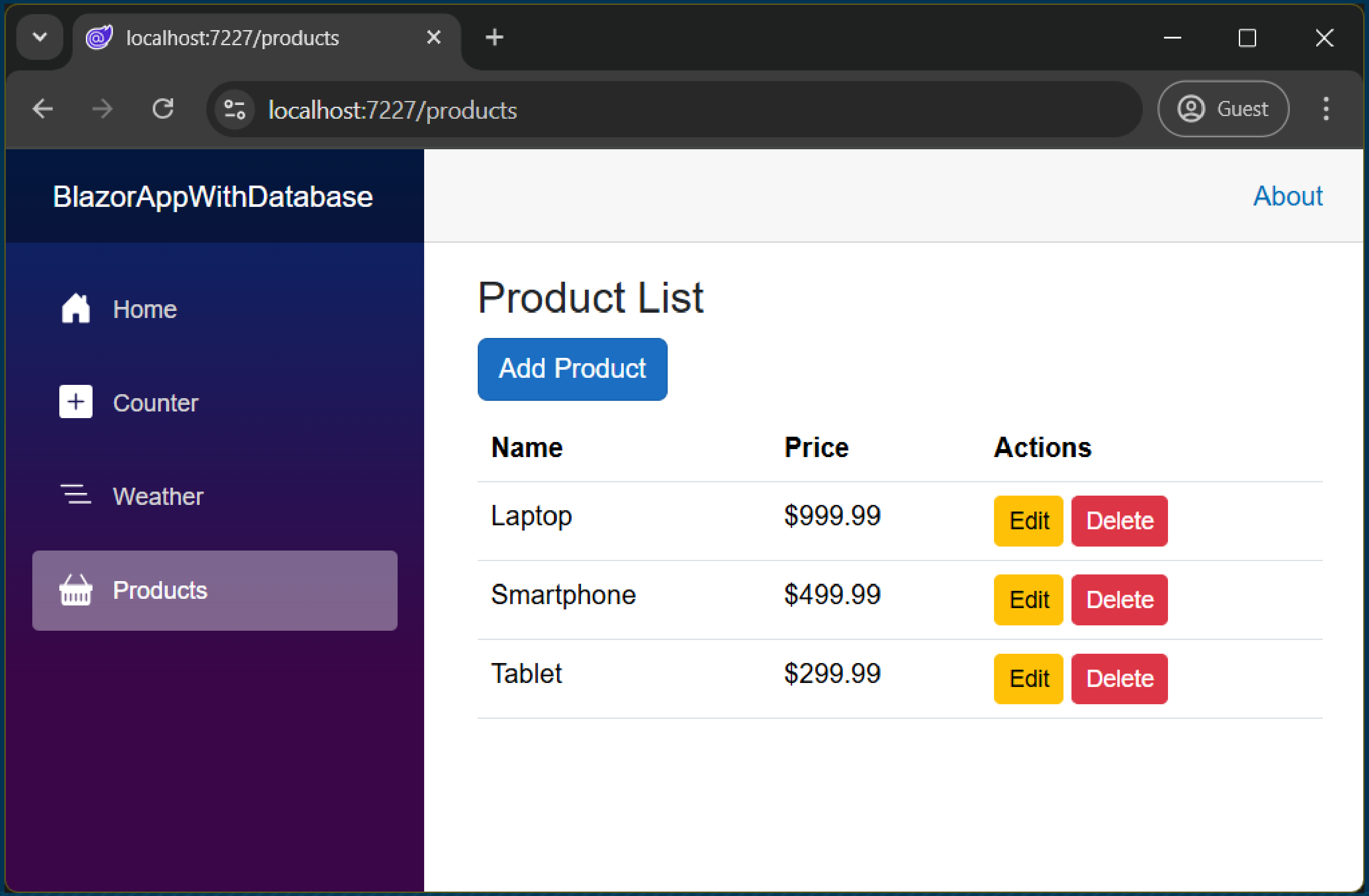


Product List

- Laptop - $999.99
- Smartphone - $499.99
- Tablet - $299.99

CRUD Operations on Products

- Modify the app to perform Add / Update / Delete functionality.



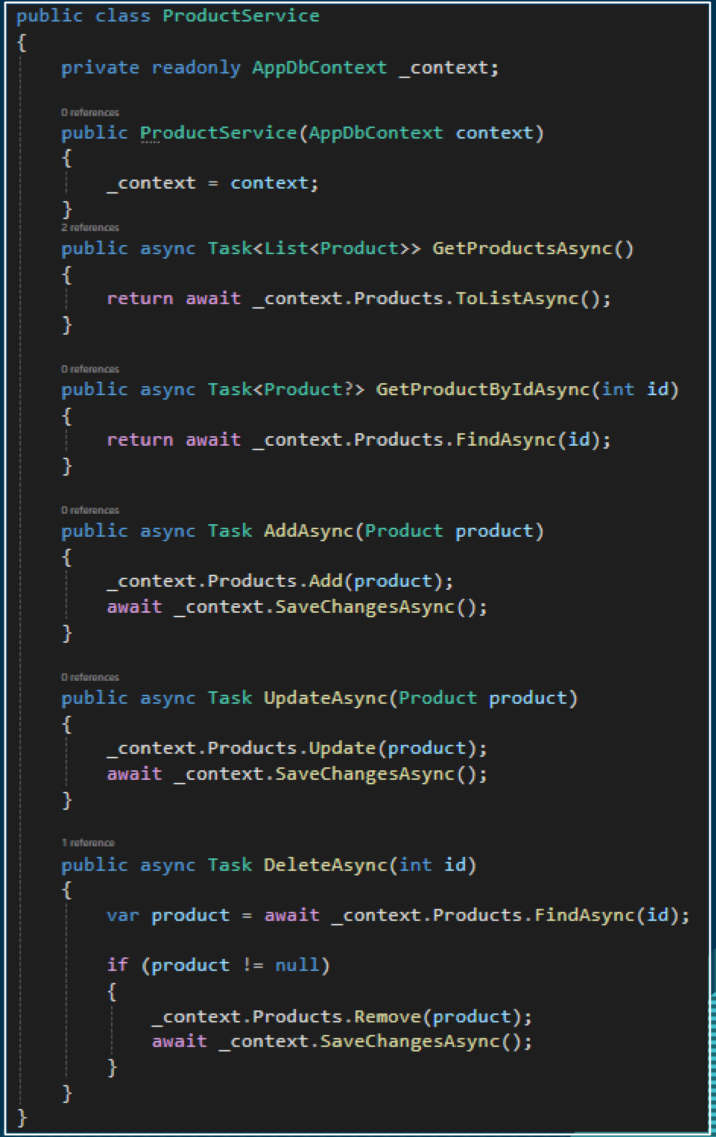
Product List

Name	Price	Actions
Laptop	$999.99	Edit Delete
Smartphone	$499.99	Edit Delete
Tablet	$299.99	Edit Delete

Update ProductService.cs

- Modify the code of the ProductService.cs and add the methods to perform Add, Update and Delete functionality.
- You can copy/paste the code from here:

```
public async Task<Product?> GetProductByIdAsync(int id)
{
    return await _context.Products.FindAsync(id);
}

public async Task AddAsync(Product product)
{
    _context.Products.Add(product);
    await _context.SaveChangesAsync();
}

public async Task UpdateAsync(Product product)
{
    _context.Products.Update(product);
    await _context.SaveChangesAsync();
}
Continue …
```

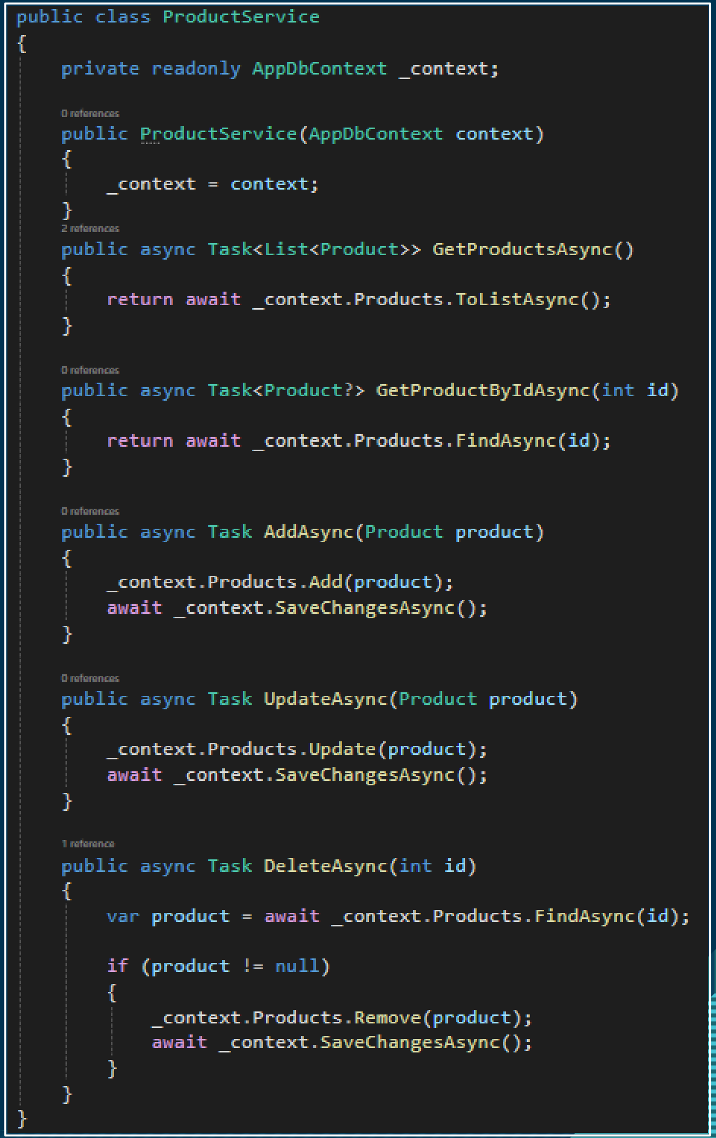


```
public class ProductService
{
    private readonly AppDbContext _context;

    0 references
    public ProductService(AppDbContext context)
    {
        _context = context;
    }
    2 references
    public async Task<List<Product>> GetProductsAsync()
    {
        return await _context.Products.ToListAsync();
    }

    0 references
    public async Task<Product?> GetProductByIdAsync(int id)
    {
        return await _context.Products.FindAsync(id);
    }

    0 references
    public async Task AddAsync(Product product)
    {
        _context.Products.Add(product);
        await _context.SaveChangesAsync();
    }

    0 references
    public async Task UpdateAsync(Product product)
    {
        _context.Products.Update(product);
        await _context.SaveChangesAsync();
    }

    1 reference
    public async Task DeleteAsync(int id)
    {
        var product = await _context.Products.FindAsync(id);

        if (product != null)
        {
            _context.Products.Remove(product);
            await _context.SaveChangesAsync();
        }
    }
}
```

Update ProductService.cs

- Modify the code of the ProductService.cs and add the methods to perform Add, Update and Delete functionality.
- You can copy/paste the code from here:

```
Continue …

public async Task DeleteAsync(int id)
{
    var product = await _context.Products.FindAsync(id);

    if (product != null)
    {
        _context.Products.Remove(product);
        await _context.SaveChangesAsync();
    }
}
```

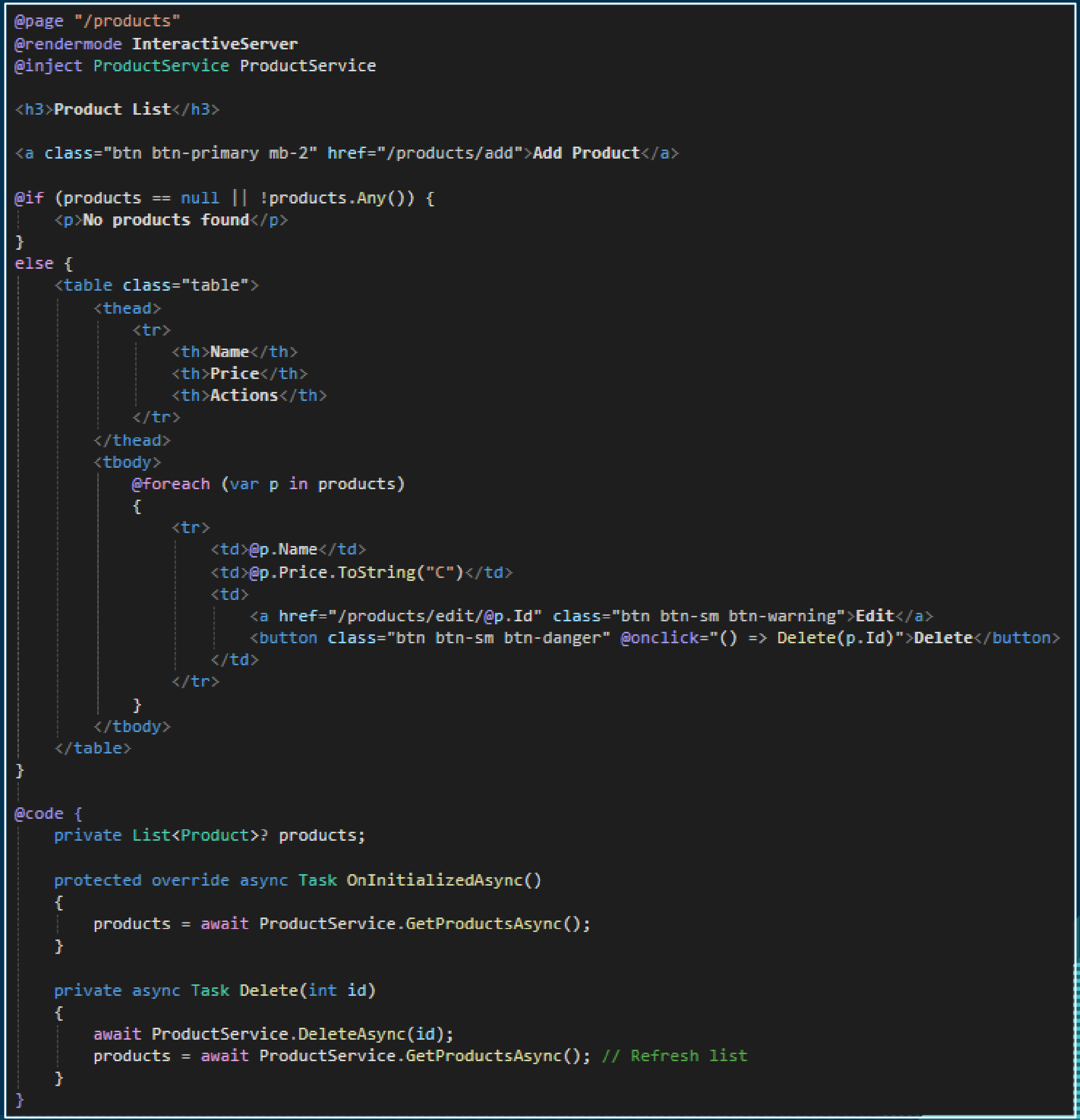


```
public class ProductService
{
    private readonly AppDbContext _context;

    0 references
    public ProductService(AppDbContext context)
    {
        _context = context;
    }
    2 references
    public async Task<List<Product>> GetProductsAsync()
    {
        return await _context.Products.ToListAsync();
    }

    0 references
    public async Task<Product?> GetProductByIdAsync(int id)
    {
        return await _context.Products.FindAsync(id);
    }

    0 references
    public async Task AddAsync(Product product)
    {
        _context.Products.Add(product);
        await _context.SaveChangesAsync();
    }

    0 references
    public async Task UpdateAsync(Product product)
    {
        _context.Products.Update(product);
        await _context.SaveChangesAsync();
    }

    1 reference
    public async Task DeleteAsync(int id)
    {
        var product = await _context.Products.FindAsync(id);

        if (product != null)
        {
            _context.Products.Remove(product);
            await _context.SaveChangesAsync();
        }
    }
}
```

Sheridan Get Creative

Update ProductList.razor

- Modify the code of the ProductList.razor to perform Add, Update and Delete functionality.
- Use Bootstrap to make it look nice.



```
@page "/products"
@rendermode InteractiveServer
@inject ProductService ProductService

<h3>Product List</h3>

<a class="btn btn-primary mb-2" href="/products/add">Add Product</a>

@if (products == null || !products.Any()) {
    <p>No products found</p>
}
else {
    <table class="table">
        <thead>
            <tr>
                <th>Name</th>
                <th>Price</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var p in products)
            {
                <tr>
                    <td>@p.Name</td>
                    <td>@p.Price.ToString("C")</td>
                    <td>
                        <a href="/products/edit/@p.Id" class="btn btn-sm btn-warning">Edit</a>
                        <button class="btn btn-sm btn-danger" @onclick="() => Delete(p.Id)">Delete</button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private List<Product>? products;

    protected override async Task OnInitializedAsync()
    {
        products = await ProductService.GetProductsAsync();
    }

    private async Task Delete(int id)
    {
        await ProductService.DeleteAsync(id);
        products = await ProductService.GetProductsAsync(); // Refresh list
    }
}
```

Update ProductList.razor

- Modify the code of the ProductList.razor to perform Add, Update and Delete functionality. You can copy/paste the code from here:

```
@page "/products"
@rendermode InteractiveServer
@inject ProductService ProductService
<h3>Product List</h3>
<a class="btn btn-primary mb-2"
href="/products/add">Add Product</a>
@if (products == null || !products.Any()) {
    <p>No products found</p>
}
else {
    <table class="table">
        <thead>
            <tr>
                <th>Name</th>
                <th>Price</th>
                <th>Actions</th>
            </tr>
        </thead>
```

```
<tbody>
            @foreach (var p in products)
            {
                <tr>
                    <td>@p.Name</td>
                    <td>@p.Price.ToString("C")</td>
                    <td>
                        <a href="/products/edit/@p.Id"
class="btn btn-sm btn-warning">Edit</a>
                        <button class="btn btn-sm btn-
danger" @onclick="() =>
Delete(p.Id)">Delete</button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
}
```

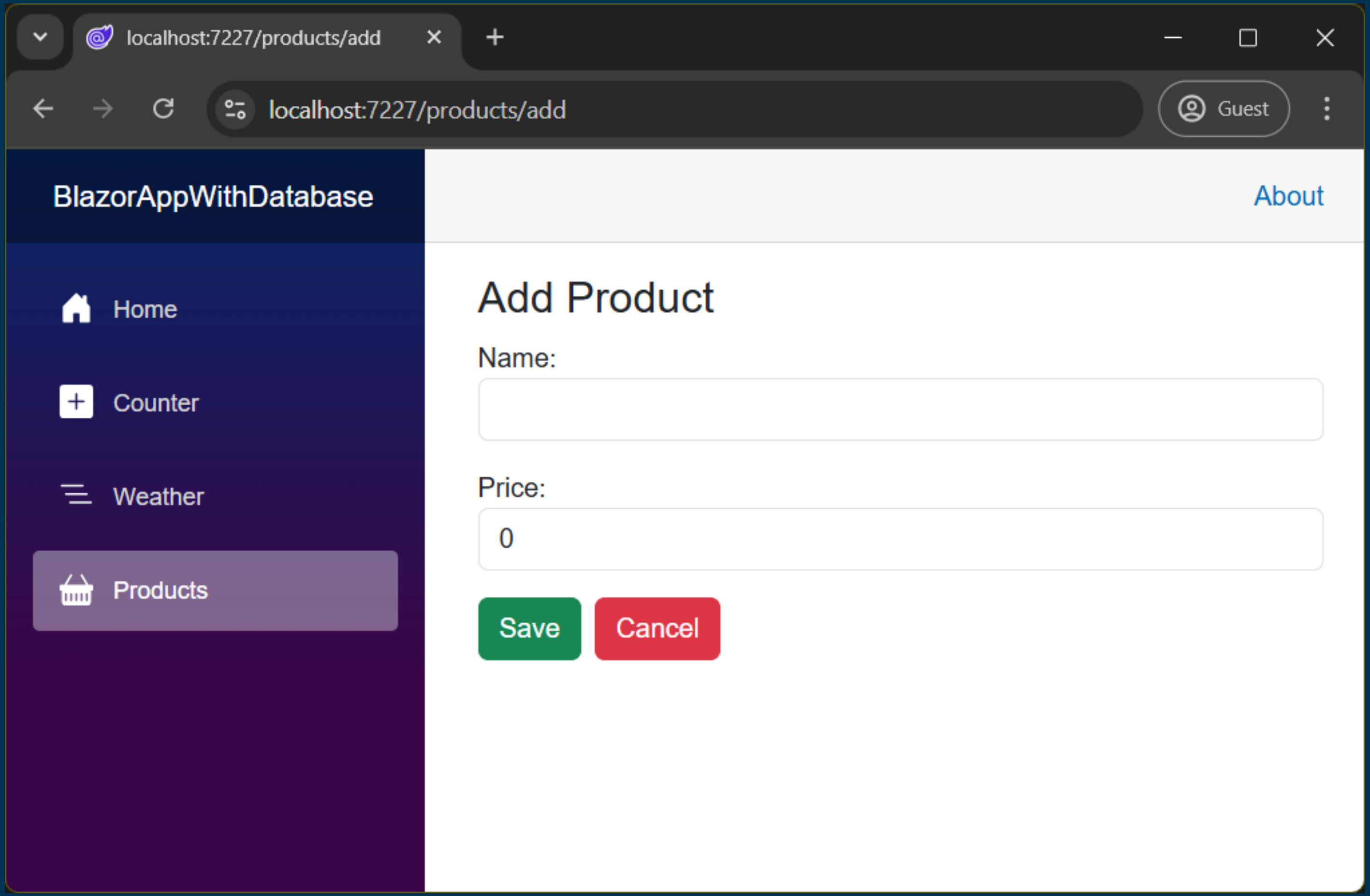
Update ProductList.razor

- Modify the code of the ProductList.razor to perform Add, Update and Delete functionality. You can copy/paste the code from here:

```
Continue …
@code {
    private List<Product>? products;

    protected override async Task
OnInitializedAsync()
    {
        products = await
ProductService.GetProductsAsync();
    }

    private async Task Delete(int id)
    {
        await ProductService.DeleteAsync(id);
        products = await
ProductService.GetProductsAsync(); // Refresh list
    }
}
```

Create AddProduct.razor

- Under Components → Pages, add a Razor Component, AddProduct.razor.



Add Product

Name:

Price:

0

Create AddProduct.razor

- Under Components → Pages, add a Razor Component, AddProduct.razor.
- You can copy/paste the code from here:

```
@page "/products/add"
@inject ProductService ProductService
@inject NavigationManager Navigation
@rendermode InteractiveServer
<h3>Add Product</h3>
<EditForm Model="product" OnValidSubmit="Save"
formname="AddProductForm">
    <div class="mb-3">
        <label>Name:</label>
        <InputText class="form-control" @bind-
Value="product.Name" />
    </div>
    <div class="mb-3">
        <label>Price:</label>
        <InputNumber class="form-control" @bind-
Value="product.Price" />
    </div>
```

```
Continue …

    <div class="d-flex gap-2">
        <button class="btn btn-success"
type="submit">Save</button>
        <button class="btn btn-danger" type="button"
@onclick="Cancel">Cancel</button>
    </div>
</EditForm>

…
```

Create AddProduct.razor

- Under Components → Pages, add a Razor Component, AddProduct.razor.
- You can copy/paste the code from here:

```
Continue …
@code {
    private Product product = new();

    private async Task Save()
    {
        await ProductService.AddAsync(product);
        Navigation.NavigateTo("/products");
    }

    private void Cancel()
    {
        Navigation.NavigateTo("/products");
    }
}
```

Explanation of AddProduct.razor

- `@inject NavigationManager Navigation`:
  - It is injected to perform programmatic page navigation (e.g., after save or cancel).

- `@rendermode InteractiveServer`:
  - Tells Blazor to render the component in Interactive Server mode.
  - Without this, events like `@onclick` and `@bind-Value` won’t work.

- `<EditForm Model="product" OnValidSubmit="Save" formname="AddProductForm">`:
  - `EditForm` is Blazor’s built-in form component that binds to a model and supports validation and submission.

  - `Model="product"`:
    - The form fields are bound to this `Product` object.

  - `OnValidSubmit="Save"`:
    - The `Save()` method will be called when the form is submitted.

  - `formname="AddProductForm"`:
    - Required in .NET 9+ to uniquely identify the form for client-side wiring.

Explanation of AddProduct.razor

- `<InputText>` and `<InputNumber>`:
  - `InputText` binds to `product.Name`.
  - `InputNumber` binds to `product.Price`.
  - `@bind-Value` enables two-way binding between the form input and the `product` object.

- Save button:
  - Submits the form (triggers `OnValidSubmit` event).

- Cancel button:
  - `type="button"` ensures this button doesn't trigger form submission.
  - `@onclick="Cancel"` triggers the `Cancel()` method.

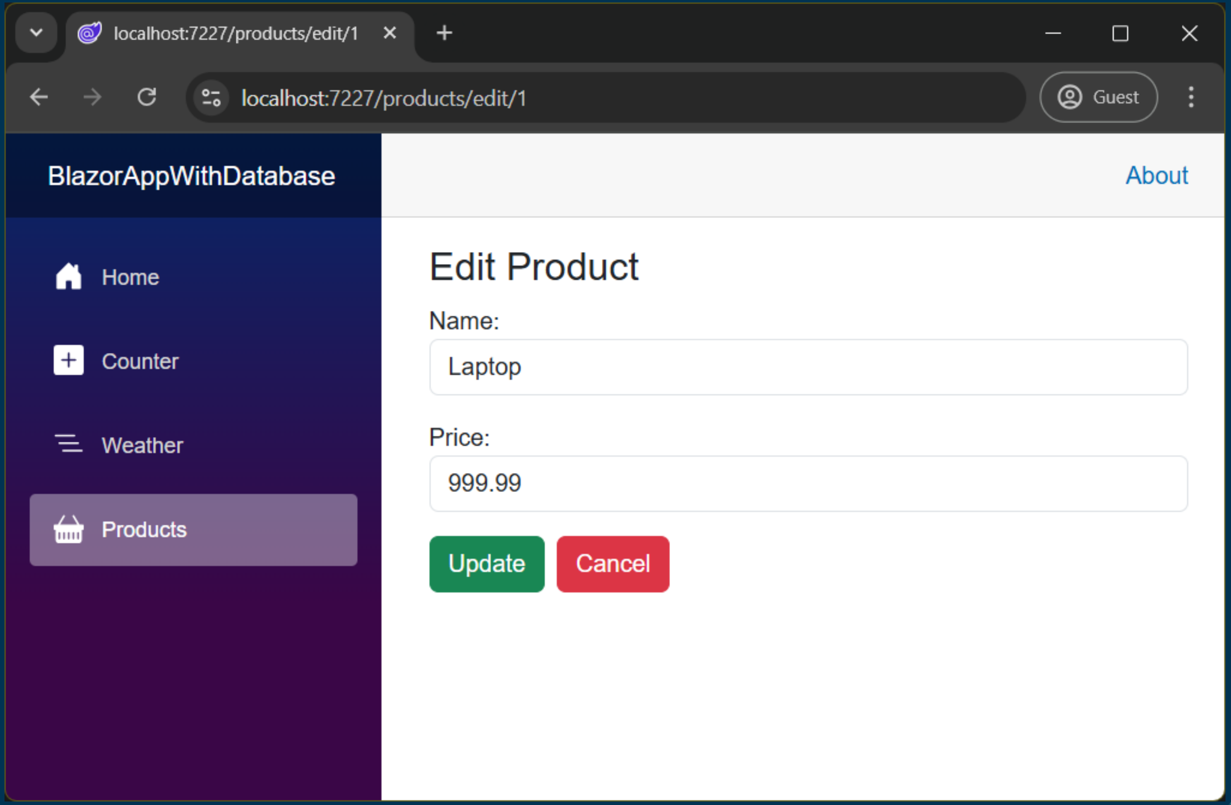
- Save method:
  - When the form is submitted, the `product` is added via the service.
  - After saving, the app navigates back to the product list.

- Cancel method:
  - Cancels the operation and takes the user back to `/products`.

Create EditProduct.razor

- Under Components → Pages, add a Razor Component, EditProduct.razor.

- `@page "/products/edit/{id:int}"`:
  - Accepts a route parameter like `/products/edit/5`.
- `[Parameter] public int Id`:
  - Binds the route `{id}` to the `Id` property.
- `ProductService.GetProductByIdAsync(Id)`:
  - Fetches the product to edit.
- `Save()`:
  - Updates the product in the database.
- `Cancel()`:
  - Navigates back without saving.


localhost:7227/products/edit/1

BlazorAppWithDatabase

About

Edit Product

Name:

Laptop

Price:

999.99

Create EditProduct.razor

- Under Components → Pages, add a Razor Component, EditProduct.razor.
- You can copy/paste the code from here:

```
@page "/products/edit/{id:int}"
@inject ProductService ProductService
@inject NavigationManager Navigation
@rendermode InteractiveServer
<h3>Edit Product</h3>
@if (product == null)
{
    <p>Product not found</p>
}

...
```

```
Continue …
else
{
   <EditForm Model="product"
OnValidSubmit="Save"
formname="EditProductForm">
        <div class="mb-3">
            <label>Name:</label>
            <InputText class="form-control" @bind-
Value="product.Name" />
        </div>

        <div class="mb-3">
            <label>Price:</label>
            <InputNumber class="form-control" @bind-
Value="product.Price" />
        </div>
…
```

Create EditProduct.razor

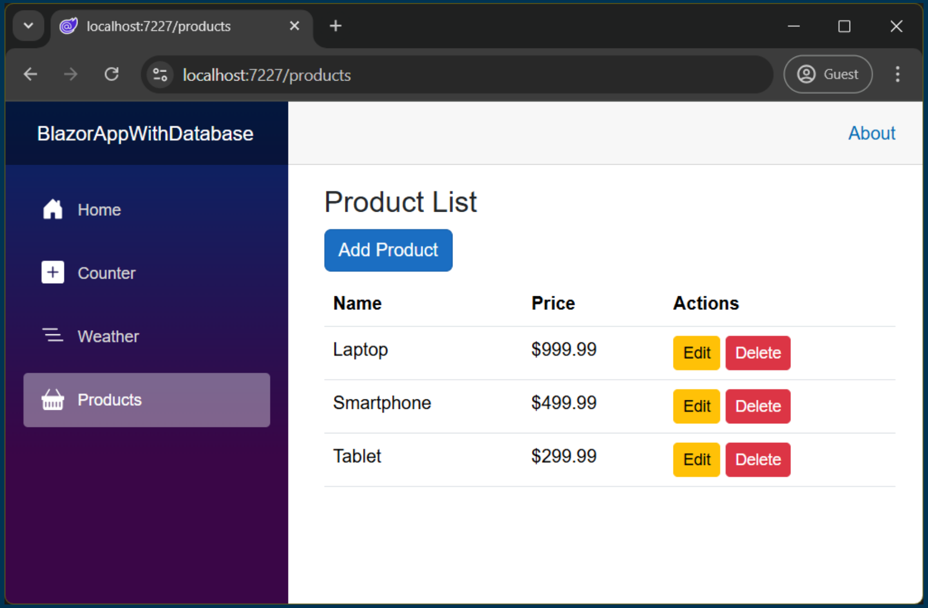
- Under Components → Pages, add a Razor Component, EditProduct.razor.
- You can copy/paste the code from here:

```
Continue …

        <div class="d-flex gap-2">
            <button class="btn btn-success"
type="submit">Update</button>
            <button class="btn btn-danger" type="button"
@onclick="Cancel">Cancel</button>
        </div>
    </EditForm>
}

@code {
    [Parameter]
    public int Id { get; set; }

    private Product? product;
```

```
Continue …
protected override async Task OnInitializedAsync() {
        product = await
ProductService.GetProductByIdAsync(Id);
        if (product == null)        {
            // Could optionally redirect or show an error
            Navigation.NavigateTo("/products");
        }        }
private async Task Save()    {
        if (product != null)  {
            await ProductService.UpdateAsync(product);
            Navigation.NavigateTo("/products");
        }        }
private void Cancel()  {
        Navigation.NavigateTo("/products");
    }
}
```

Delete Functionality in ProductList.razor

- The delete functionality allows users to remove a product from the list.
- Once deleted, the product is removed from the UI by re-fetching the list.



Product List

Name	Price	Actions
Laptop	$999.99	Edit Delete
Smartphone	$499.99	Edit Delete
Tablet	$299.99	Edit Delete

```
<button class="btn btn-sm btn-danger" @onclick="() => Delete(p.Id)">Delete</button>
```

- Adds a red Delete button for each product.
- When clicked, it triggers the `Delete()` method and passes in the ID of that product (`p.Id`).

Do It Yourself!

- **Display Filtered Records by Category**:
  - Create a `Category` property in the `Product` model (e.g., "Electronics", "Clothing", etc.).
  - Add a dropdown list on the `/products` page with all distinct categories.
  - When a category is selected, filter the product list to only show products in that category.

🛠 Do It Yourself!

- **Track and Display Recently Added Products**:
  - Add a `DateAdded` property to the `Product` model.
  - On the homepage, display the 5 most recently added products, ordered by `DateAdded DESC`.

🛠️ Do It Yourself!

- **Create a Basic Search Function**:
  - Add a search box at the top of the product list page.
  - As the user types a name (e.g., "Phone"), filter and display only products whose names contain the search text.

Thank You

Copyright

Sheridan Get Creative