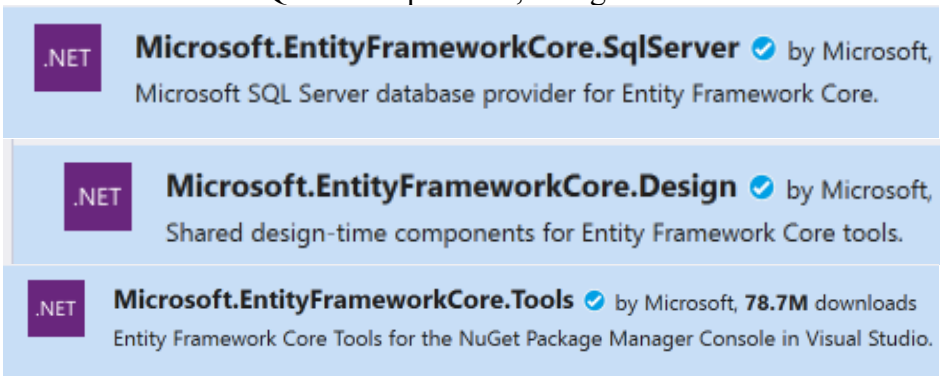
Using Entity Framework (And Scaffolding)
.NET Web Application

Entity Framework is useful when our data is stored in a database and we want to use this data in our .NET Application. It allows us to connect to databases and manipulate our data without writing much database code.

Create an empty Web project.
Same as we did with the Console app, we can include the EF packages.

Solution Explorer → Manage NuGet Packages

We will install the SQL Server provider, Design and Core Tools.



Create Data folder and create the context.
Add a constructor with options.

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using WebFruitStore.Models;

namespace WebFruitStore.Data
{
    public class FruitStoreContext : DbContext
    {
        //Properties (database tables)
        public DbSet <Customer> Customers { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<Product> Products { get; set; }
        public DbSet<ProductOrder> ProductOrders { get; set; }

        // Constructor
        public FruitStoreContext(DbContextOptions options) : base(options)
        {
        }
    }
}
```

Create Entity Model

Create Models folder and create entity models.
Do this the same as we did for the Console application.

Product.cs

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace FruitStore.Models
{
    public class Product
    {
        // Proerties

        // Id or ProductID is the Key [Key]
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }

        [Column(TypeName = "decimal(18,2)")]
        public decimal Price { get; set; }
    }
}
```

Customer.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace FruitStore.Models
{
    public class Customer
    {
        // Properties
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        // Navigation property (a customer can have zero or more orders)
        public ICollection<Order> Orders { get; set; }

    }
}
```

Order.cs

```
using System;
using System.Collections.Generic;

namespace FruitStore.Models
{
    public class Order
    {
        // Properties
        public int ID { get; set; }
        public DateTime OrderDate { get; set; }
        public DateTime? OrderCompleted { get; set; }

        // Foreign key relationship to the Customer table
        public int CustomerID { get; set; }

        // Navigation properties (one customer per order)
        public Customer Customer { get; set; }
        // Navigation properties (to out intersection table)
        public ICollection<ProductOrder> ProductOrders { get; set; }
    }
}
```

ProductOrder.cs (intersection table)

```
namespace FruitStore.Models
{
    public class ProductOrder
    {
        // Properties
        public int Id { get; set; }
        public int Quantity { get; set; }

        // Represent the foreign key relationships
        public int OrderID { get; set; }
        public int ProductID { get; set; }

        // Navigation properties (order and product)
        public Order Order { get; set; }
        public Product Product { get; set; }
    }
}
```

Register the DbContext

Register the context with ASP .NET's Dependency Injection container.
We are hard-coding the path to an existing database for now.

Startup.cs

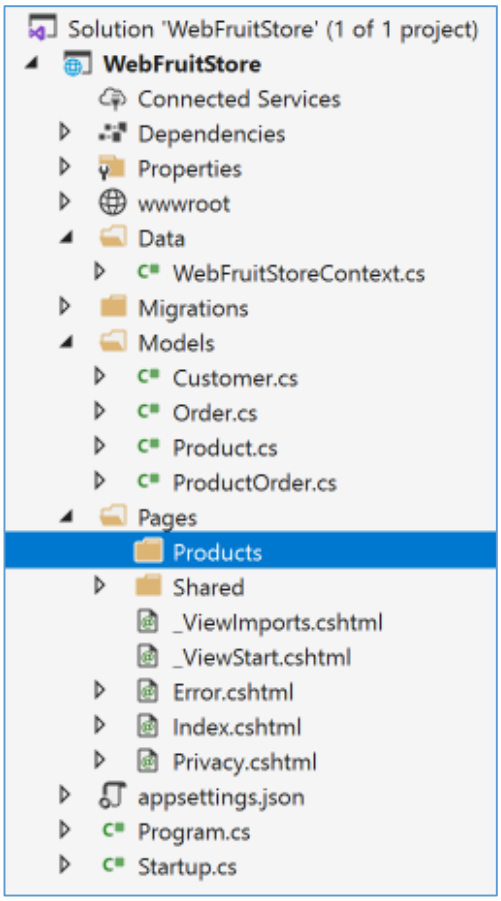
```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<WebFruitStoreContext>(options =>
    options.UseSqlServer("Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog=FruitStoreContext; Integrated Security=True; ConnectRetryCount=0"));
}
```

```
services.AddDbContext<FruitStoreContext>(options =>
            options.UseSqlServer("Data Source=(localdb)\\MSSQLLocalDB; Initial
Catalog=FruitStoreContext; Integrated Security=True; ConnectRetryCount=0"));
```

Next Steps:

- Understand connection strings and how to implement them correctly in the real world.

Creating New Pages (using Scaffolding)



Add a Products subfolder under Pages.
We are going to create new pages to create, read, update and delete products (CRUD).

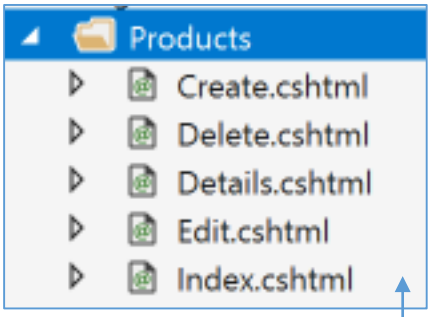
Add → New Razor Page → Razor Pages using Entity Framework (CRUD)

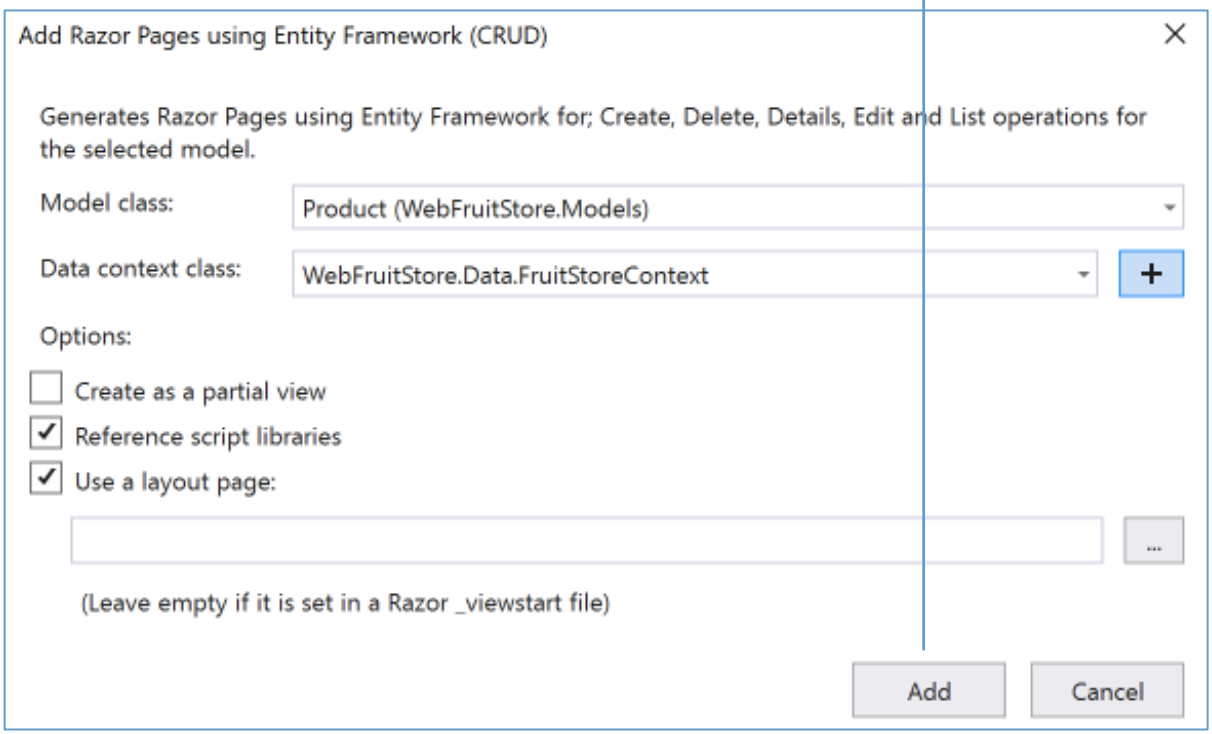
Select Product as the model, and also select the data context by clicking on the Plus button.

Check the data context and modify it if required.

When you click [Add], Visual Studio will create a set of pages inside the Products folder.

These pages are created for us:





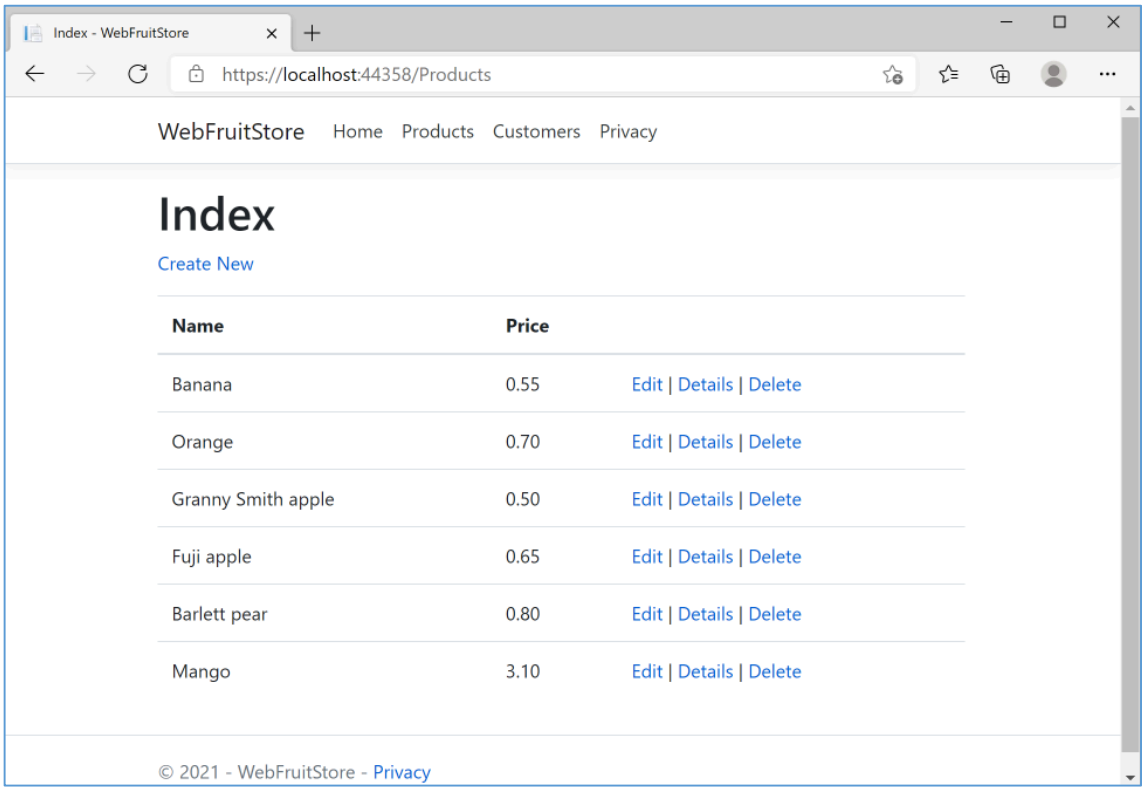
Viewing The New Pages (created using Scaffolding)

Run the application and change the URL to /Products
This will get the Products/Index.cshtml page and will display it in the browser.

You can see that
Index.cshtml displays the products from the Products table in the database.
Create.cshtml allows you to create new products and save to the database.
Edit.cshtml allows you to edit the data stored for the selected product.
Delete.cshtml allows you to delete the selected product from the database.

Next Steps:

- Understanding the URLs used by these pages and how the product ID is used.
- Understanding the HTML and C# code that was generated behind these pages.



Understanding The Products Index Page (List Of Products)

Index.cshtml

A HTML table has been created with the list of products.
The name and price are displayed for each product.
There is also a link to Edit, Details and Delete.

```
@page
@model WebFruitStore.Pages.Products.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Product[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Product[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Product) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

In its code-behind (buddy C# page), the constructor handles the database context (session) to connect to the database. We are injecting the context into the constructor (dependency injection does this for us, we just have to create the constructor).

The page obtains the list of products, storing them in the Product list property.
This is what is used by the cshtml page (the view) in the foreach loop.

Index.cshtml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using WebFruitStore.Data;
using WebFruitStore.Models;

namespace WebFruitStore.Pages.Products
{
    public class IndexModel : PageModel
    {
        private readonly WebFruitStore.Data.FruitStoreContext _context;

        public IndexModel(WebFruitStore.Data.FruitStoreContext context)
        {
            _context = context;
        }

        public IList<Product> Product { get;set; }

        public async Task OnGetAsync()
        {
            Product = await _context.Products.ToListAsync();
        }
    }
}
```

We can click on the Create New link and add a new product using the form provided.
We can also view the details for a selected product, or edit or delete a product.
All the working code for these pages was created for us using Scaffolding.

Understanding The Individual Product Page (View Product)

Details.cshtml

This page displays the product selected using HTMLHelpers.
It shows the name and price (as these are the fields in the product database table).

You can see the product ID in the URL of the page.

```
@page
@model WebFruitStore.Pages.Products.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Product</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Product.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Product.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Product.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Product.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Product.Id">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>

```

Next Steps:

- Understanding what HTMLHelpers are and how to use them.

In its code-behind (buddy C# page), the constructor handles the database context (session) to connect to the database. The page gets a single product based on the ID.

This product is then displayed by the page.

Details.cshtml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using WebFruitStore.Data;
using WebFruitStore.Models;

namespace WebFruitStore.Pages.Products
{
    public class DetailsModel : PageModel
    {
        private readonly WebFruitStore.Data.FruitStoreContext _context;

        public DetailsModel(WebFruitStore.Data.FruitStoreContext context)
        {
            _context = context;
        }

        public Product Product { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Product = await _context.Products.FirstOrDefaultAsync(m => m.Id == id);

            if (Product == null)
            {
                return NotFound();
            }
            return Page();
        }
    }
}
```

Understanding The Create Page (Adding A New Product)

We can click on the Create New link on the Index page and add a new product using the form provided.

The OnGet() method for the Create.cshtml page returns the empty form (this is a Razor view). This contains the HTML form elements, labels and validation which enforces the constraints we included in our Entity Model.

When we post the form, the form fields are bound to the properties in our Model, which are then saved to the database table. The user is then redirected back to Index.cshtml/

Create.cshtml

```
@page
@model WebFruitStore.Pages.Products.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Product</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Product.Name" class="control-label"></label>
                <input asp-for="Product.Name" class="form-control" />
                <span asp-validation-for="Product.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Product.Price" class="control-label"></label>
                <input asp-for="Product.Price" class="form-control" />
                <span asp-validation-for="Product.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Create.cshtml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using WebFruitStore.Data;
using WebFruitStore.Models;

namespace WebFruitStore.Pages.Products
{
    public class CreateModel : PageModel
    {
        private readonly WebFruitStore.Data.FruitStoreContext _context;

        public CreateModel(WebFruitStore.Data.FruitStoreContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Product Product { get; set; }

        // To protect from overposting attacks, see https://aka.ms/RazorPagesCRUD
        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Products.Add(Product);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}

```

Understanding The Edit Page
(To Edit A Product)

When we click on the Edit link, a form appears (pre-populated with data from the table). This form is created using tag helpers.

Edit.cshtml

```
@page
@model WebFruitStore.Pages.Products.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Product</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Product.Id" />
            <div class="form-group">
                <label asp-for="Product.Name" class="control-label"></label>
                <input asp-for="Product.Name" class="form-control" />
                <span asp-validation-for="Product.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Product.Price" class="control-label"></label>
                <input asp-for="Product.Price" class="form-control" />
                <span asp-validation-for="Product.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Next Steps:

- Understanding tag helpers for forms in Razor pages.

The OnGet method queries the products in the database table for the product that matches the ID that was passed in with the URL. The product is retrieved and presented for user to edit.

When the user posts the form, the model binder grabs the product object and attaches it to the dbcontext, finds the existing product, marks it as modified and saves the changes made.

Edit.cshtml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using WebFruitStore.Data;
using WebFruitStore.Models;

namespace WebFruitStore.Pages.Products
{
    public class EditModel : PageModel
    {
        private readonly WebFruitStore.Data.FruitStoreContext _context;

        public EditModel(WebFruitStore.Data.FruitStoreContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Product Product { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Product = await _context.Products.FirstOrDefaultAsync(m => m.Id == id);

            if (Product == null)
            {
                return NotFound();
            }
            return Page();
        }

        // To protect from overposting attacks, enable the specific properties you want to bind to.
        // For more details, see https://aka.ms/RazorPagesCRUD.
        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Attach(Product).State = EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!ProductExists(Product.Id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }

            return RedirectToPage("./Index");
        }

        private bool ProductExists(int id)
        {
            return _context.Products.Any(e => e.Id == id);
        }
    }
}
```

Understanding The Delete Page
(To Delete A Product)

This is similar to the Edit page.

The OnGet method will retrieve the product matching the ID and display it for the user.

When the user clicks to Post, we find the same product, call Remove on the product table and save the changes.

Delete.cshtml

```
@page
@model WebFruitStore.Pages.Products.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h1>Delete</h1>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Product</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Product.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Product.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Product.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Product.Price)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Product.Id" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Delete.cshtml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using WebFruitStore.Data;
using WebFruitStore.Models;

namespace WebFruitStore.Pages.Products
{
    public class DeleteModel : PageModel
    {
        private readonly WebFruitStore.Data.FruitStoreContext _context;

        public DeleteModel(WebFruitStore.Data.FruitStoreContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Product Product { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Product = await _context.Products.FirstOrDefaultAsync(m => m.Id == id);

            if (Product == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Product = await _context.Products.FindAsync(id);

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

            return RedirectToPage("./Index");
        }
    }
}
```