Routing and Layouts in ASP.NET Blazor

What is Routing in Blazor?

- Routing in Blazor enables navigation between different Razor components based on the URL.
- When a user navigates to a URL, Blazor finds the component with the matching `@page` directive and renders it.
- Blazor supports client-side routing, which means the page doesn't reload, rather the navigation is handled within the browser using the `Router` component.

@page Directive

- Each routable component (like a Razor page) must begin with an `@page` directive, defining the route it responds to.

```
@page "/about"
```

- This means that the About component will be displayed when the user navigates to `/about`.

App.razor and the <Router> Component

- App.razor acts as the root component for the Blazor app.
- It contains the `<Routes />` component.
- Routes.razor has `<Router>` element which:
  - Matches the current URL to a route.
  - Displays the appropriate component.
  - If you want to customize the fallback behavior, you can use the `NotFound`.

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Routes.razor

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

- Routes.razor:
  - Uses `<Router>` to match routes.
  - Displays pages using `<RouteView>`.
  - Shows a custom 404 message when no route matches using `<NotFound>`.

Exercise - Routing Basics

- Create a Blazor app that will have 3 pages:
  - Home (`/`)
  - About (`/about`)
  - Contact (`/contact`)

- The Home.razor already has the `@page` directive.

```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.
```

Exercise - Routing Basics

- Add a new razor component About.razor.
- Add the @page directive at the top.
- Add some HTML tags.

```
@page "/about"

<h1>About Us</h1>

<p>This is a sample Blazor application for learning routing.</p>
```

Exercise - Routing Basics

- Open Layout/NavMenu.razor.
- Add navigation using `NavLink`.
- Blazor's `NavLink` automatically adds the active class when the link matches the current route.

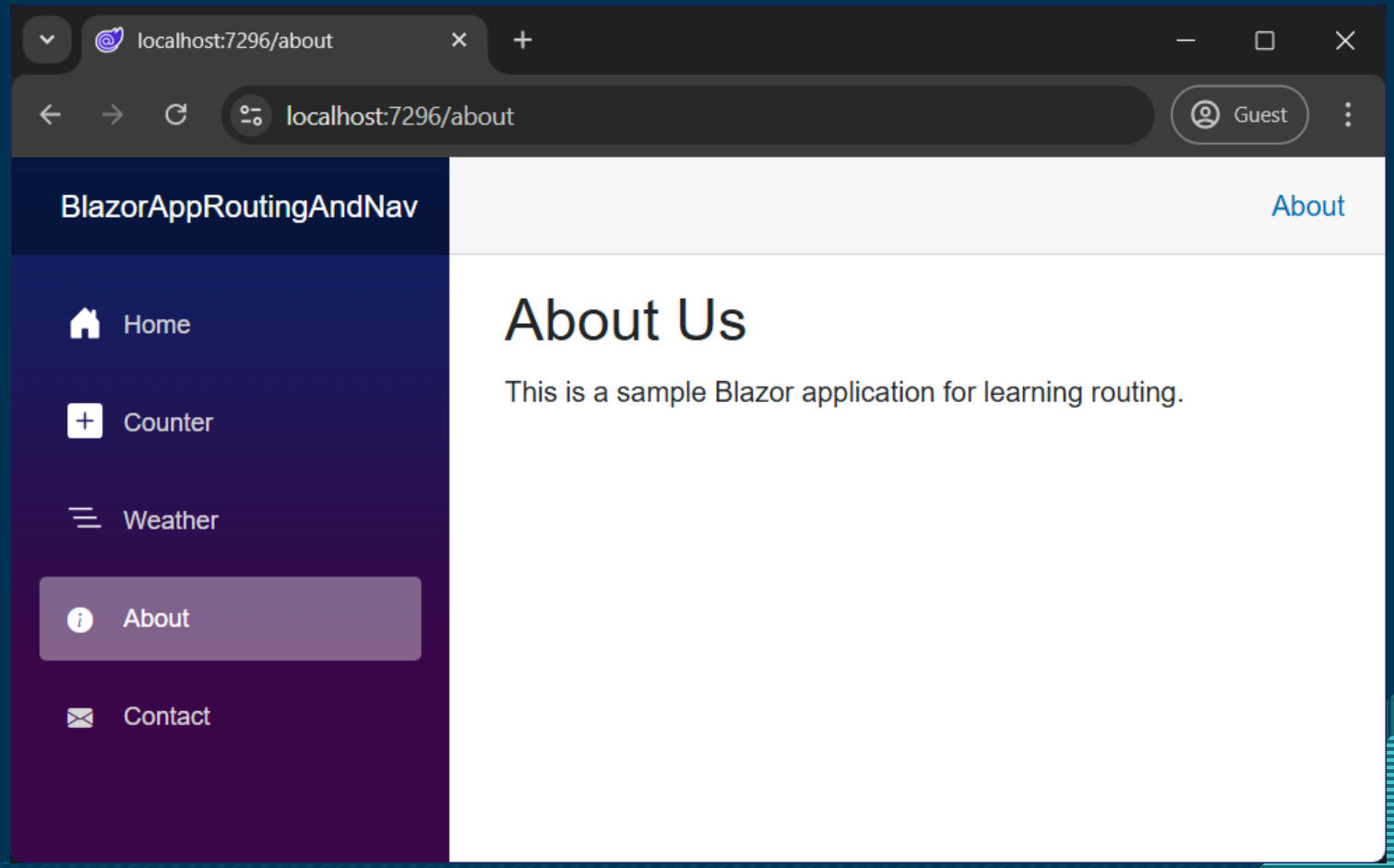
```
    <div class="nav-item px-3">
        <NavLink class="nav-link" href="weather">
            <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Weather
        </NavLink>
    </div>

    <div class="nav-item px-3">
        <NavLink class="nav-link" href="about">
            <span class="bi bi-info-circle-fill mb-4" aria-hidden="true"></span> About
        </NavLink>
    </div>

    <div class="nav-item px-3">
        <NavLink class="nav-link" href="contact">
            <span class="bi bi-envelope-fill mb-4" aria-hidden="true"></span> Contact
        </NavLink>
    </div>
</nav>
```

- Add the following CSS to the App.razor's `<head>` element.

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.13.1/font/bootstrap-icons.css" rel="stylesheet">
```

Exercise - Routing Basics

- Run the app.
- Click the navigation links.
- Enter URLs directly (e.g., `/about`, `/contact`).



Route Parameters in Blazor

- Route parameters allow a Blazor component to accept values from the URL.
- For example, if you navigate to:

```
/product/123
```

- You can have a Blazor component (like Product.razor) that reads 123 as a parameter (`id`).
- This enables dynamic routing, where a single component can handle multiple data entries, like user profiles, orders, blog posts, etc.

Basic Route Parameter

- Example: User.razor:

```
@page "/user/{id}"

<h3>User ID: @Id</h3>

@code {
    [Parameter]
    public string? Id { get; set; }
}
```

- The `@page "/user/{id}"` tells Blazor to match any route like `/user/123`, `/user/abc` etc.
- The `id` value in the URL is passed to the `Id` property decorated with `[Parameter]`.
- You can use any variable name inside `{}`, just make sure it matches the `[Parameter]` property name.
  - It is case-insensitive, but spellings must match.

Typed Route Parameters

- Blazor supports type constraints like `int`, `guid`, `bool`, etc.
- Example: Order.razor:

```
@page "/order/{orderId:int}"

<h3>Order ID: @OrderId</h3>

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

- Works for:
  - `/order/100`
  - `/order/9999`
- Fails for:
  - `/order/abc` → Won't match.

Multiple Parameters

- You can pass multiple route parameters.
- Example: Invoice.razor:

```
@page "/invoice/{year:int}/{month:int}"

<h3>Invoice for @Year/@Month</h3>

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

    [Parameter]
    public int Month { get; set; }
}
```

- Works for:
  - `/invoice/2024/12`

Optional Parameters (Workaround)

- Blazor doesn't support optional route parameters directly in the `@page` directive.
- But you can define multiple routes for the same component.
- Example: Search.razor:

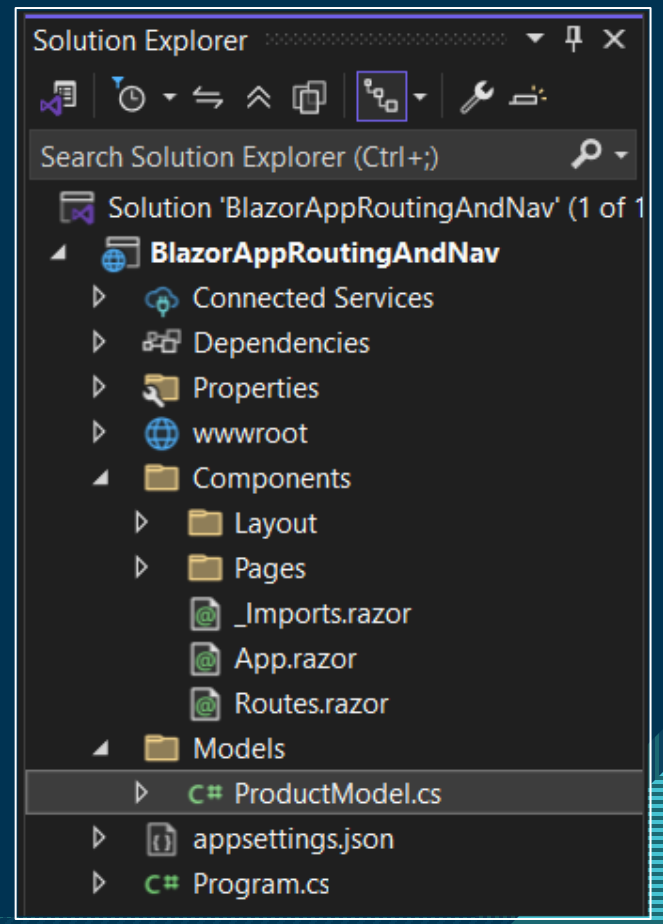
```
@page "/search"
@page "/search/{term}"

<h3>Search Results</h3>
@if (Term != null)
{
    <p>Results for: <strong>@Term</strong></p>
}
else
{
    <p>Please enter a search term.</p>
}

@code {
    [Parameter]
    public string? Term { get; set; }
}
```

- Works for:
  - `/search`
  - `/search/books`

Exercise - Route Parameter

- Create a new folder Models at the root of your project.
- Create a new class ProductModel.cs inside the Models folder.

```
public class ProductModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }
}
```



Exercise - Route Parameter

- Open _Imports.razor and add the path to the Models folder.
- Please change the project name "BlazorAppRoutingAndNav" to your own 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 BlazorAppRoutingAndNav
@using BlazorAppRoutingAndNav.Components
@using BlazorAppRoutingAndNav.Models
```

Exercise - Route Parameter

- Create a new component, Product.razor:
  - You can copy the sample code from here:

```
@page "/product/{id:int}"
<h1>Product Details</h1>
@if (product != null) {
    <div class="card p-3 m-2 shadow">
        <h4>@product.Name</h4>
        <p><strong>ID:</strong>
@product.Id</p>
        <p><strong>Price:</strong>
@product.Price.ToString("C")</p>
    </div>          }
Else {
    <div class="alert alert-warning">
Product not found.<NavLink href="/">Go
back to home</NavLink> </div>
}
```

```
@code {
    [Parameter]
    public int Id { get; set; }
    private ProductModel? product;
    private List<ProductModel>
products = new() {
        new ProductModel(1, "Gaming
Laptop", 1499.99M),
        new ProductModel(2, "Wireless
Mouse", 29.99M),
        new ProductModel(3,
"Mechanical Keyboard", 79.50M),
        new ProductModel(4, "27-inch
Monitor", 299.00M),
        new ProductModel(5, "USB-C
Dock", 59.95M)     };
```

Exercise - Route Parameter

- Create a new component, Product.razor:
  - You can copy the sample code from here:

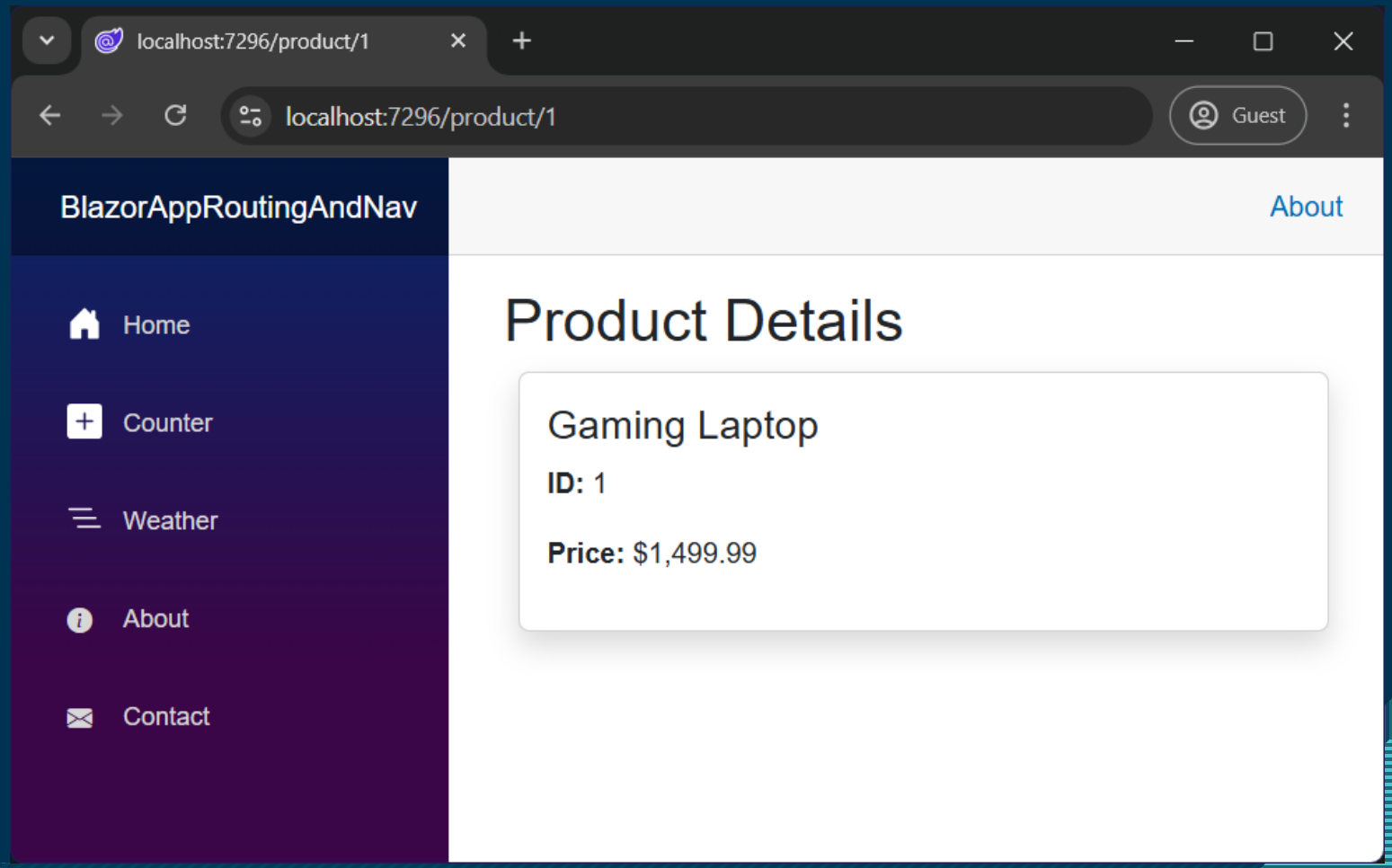
```
Continue …
    protected override void OnInitialized()
{
        product = products.Find(p => p.Id
== Id); }
}
```

```
protected override void OnInitialized()
{
    product = products.Find(p => p.Id == Id);
}
```

- The `OnInitialized` method is a component lifecycle method that runs once when the component is first initialized (before rendering for the first time).
- This method is called when the `Product` component is created.
- It tries to find a `product` in the `products` list whose `Id` matches the `Id` parameter.
- The result is stored in the `product` variable, which is then used to display product details.

Exercise - Route Parameter

- Run the app and try navigating to:
  - `/product/1`
  - `/product/4`
  - `/product/99` (not found)



Layout Components

What Are Layout Components?

- Layout Components are like master pages or templates in traditional web apps.
- They define a common structure for your pages, such as headers, footers, nav menus, and sidebars.
- And then insert page-specific content inside that layout.

- Why Use Layouts?
  - Reuse consistent UI structure (e.g., navigation and footer).
  - Keep pages clean and focused only on their content.
  - Centralized styling and branding.
  - Easy to change layout globally (e.g., update nav for all pages).

Anatomy of a Layout Component

- A layout is a Razor component with a special directive `@inherits LayoutComponentBase`.
- It must contain a placeholder for the content of child pages using `@Body`.

- Have a look at the Layout/MainLayout.razor.

- `@inherits LayoutComponentBase` tells the Razor component to inherit behavior from the built-in class `LayoutComponentBase`, which is part of the Blazor framework.
- This is required when creating a layout component because it provides the special Blazor layout functionality, specifically, access to the `@Body` property.

MainLayout.razor

BlazorAppRoutingAndNav

```
1   @inherits LayoutComponentBase
2
3   <div class="page">
4       <div class="sidebar">
5           <NavMenu />
6       </div>
7
8       <main>
9           <div class="top-row px-4">
10              <a href="https://learn.microsoft.com/aspnet/core/" target="_blank">About</a>
11          </div>
12
13          <article class="content px-4">
14              @Body
15          </article>
16      </main>
17  </div>
18
19  <div id="blazor-error-ui" data-nosnippet>
20      An unhandled error has occurred.
21      <a href="." class="reload">Reload</a>
22      <span class="dismiss">✕</span>
23  </div>
```

Register Layout in the Router (Routes.razor)

- In Routes.razor, `<RouteView>` defines the default layout.
- This applies `MainLayout` to all routed components unless they override it with `@layout`.

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(Layout.MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1" />
    </Found>
</Router>
```

Exercise - Layout Component

- Create a new Razor component file: Layout/MyLayout.razor.
  - You can copy-paste the code from here:

```
@inherits LayoutComponentBase

<div class="container-fluid">
    <header class="bg-primary text-white p-3">
        <h1>My Blazor App</h1>
    </header>

    <div class="row">
        <div class="col-4 bg-dark p-2">
            <NavMenu />
        </div>

Continue …
```

```
Continue …

        <div class="col-8 p-3">
            @Body
        </div>
    </div>

    <footer class="mt-4 p-3 text-center text-muted">
        © 2025 My Company
    </footer>
</div>
```

Exercise - Layout Component

- Create a new Razor component file: Layout/MyLayout.razor.

- @Body is where the content of the current page will be rendered.
- You can style it however you like using Bootstrap or custom CSS.

MyLayout.razor

```
@inherits LayoutComponentBase

<div class="container-fluid">
    <header class="bg-primary text-white p-3">
        <h1>My Blazor App</h1>
    </header>

    <div class="row">
        <div class="col-4 bg-dark p-2">
            <NavMenu />
        </div>

        <div class="col-8 p-3">
            @Body
        </div>
    </div>

    <footer class="mt-4 p-3 text-center text-muted">
        © 2025 My Company
    </footer>
</div>
```

Exercise - Layout Component

- Add the path to the Layout folder in the _Imports.razor.
- Change the project name "BlazorAppRoutingAndNav" to your own project name.

_Imports.razor

```
gAndNav
@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 BlazorAppRoutingAndNav
@using BlazorAppRoutingAndNav.Components
@using BlazorAppRoutingAndNav.Components.Layout
@using BlazorAppRoutingAndNav.Models
```

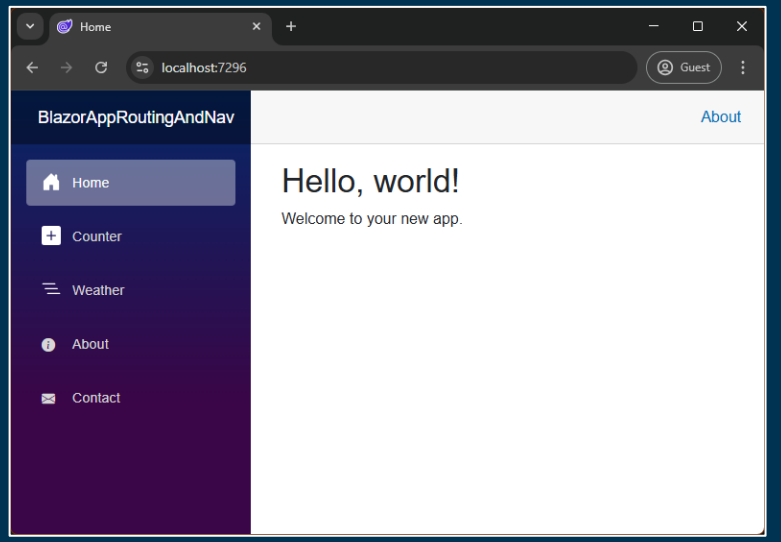
Exercise - Layout Component

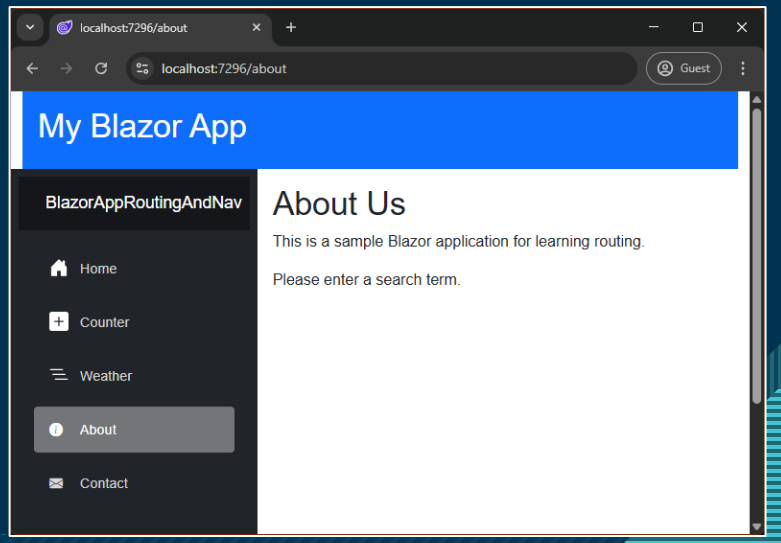
- Apply `MyLayout` to the `About.razor`.

About.razor | MyLayout.razor

ndNav | BlazorAppRoutingAndNav.Components.Pages.About | Id

```
@page "/about"
@page "/about/{id:int}"

@layout MyLayout

<h1>About Us</h1>

<p>This is a sample Blazor application for learning routing.</p>
```





Nested Layouts

- A nested layout is when one layout component is wrapped inside another layout.
- Blazor allows components to use another layout as their layout using: `@layout BaseLayout`.

- So, a layout (like `AdminLayout`) can:
  - Act as a layout for pages.
  - Also be wrapped inside another layout (like `BaseLayout`).
- This creates a layout hierarchy, where each level adds structure.

- Think of `BaseLayout` as the outer HTML frame (like the `<html>` and `<body>`), and `MainLayout` or `AdminLayout` as sections inside it (like sidebar + top row).

```
BaseLayout
├── MainLayout
│    └── Home.razor, About.razor, Product.razor
├── AdminLayout
     └── AdminProduct.razor, AdminDashboard.razor
```

Why Use Nested Layouts?

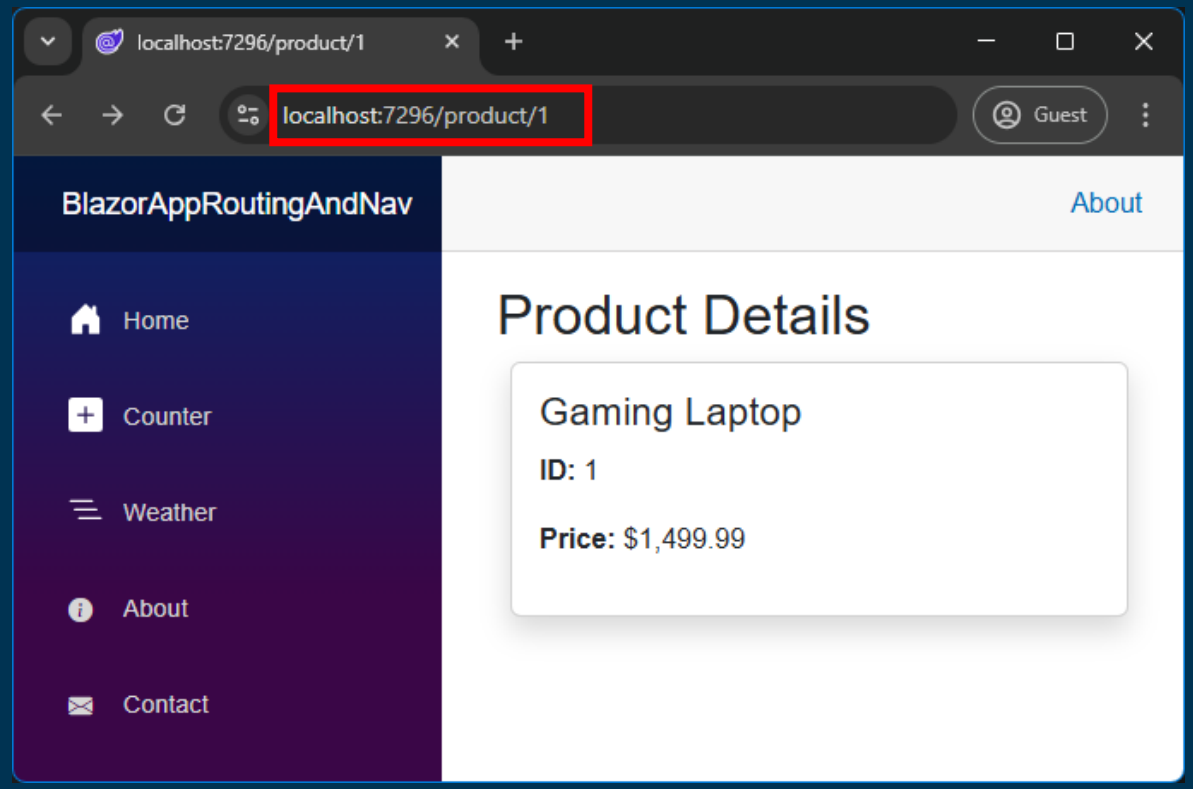
- **Avoids Code Duplication**:
  - Without nesting: MainLayout and AdminLayout would both define the same , error UI, etc.
  - With nesting, you put shared UI (like the error banner) in BaseLayout, and both layouts inherit it.

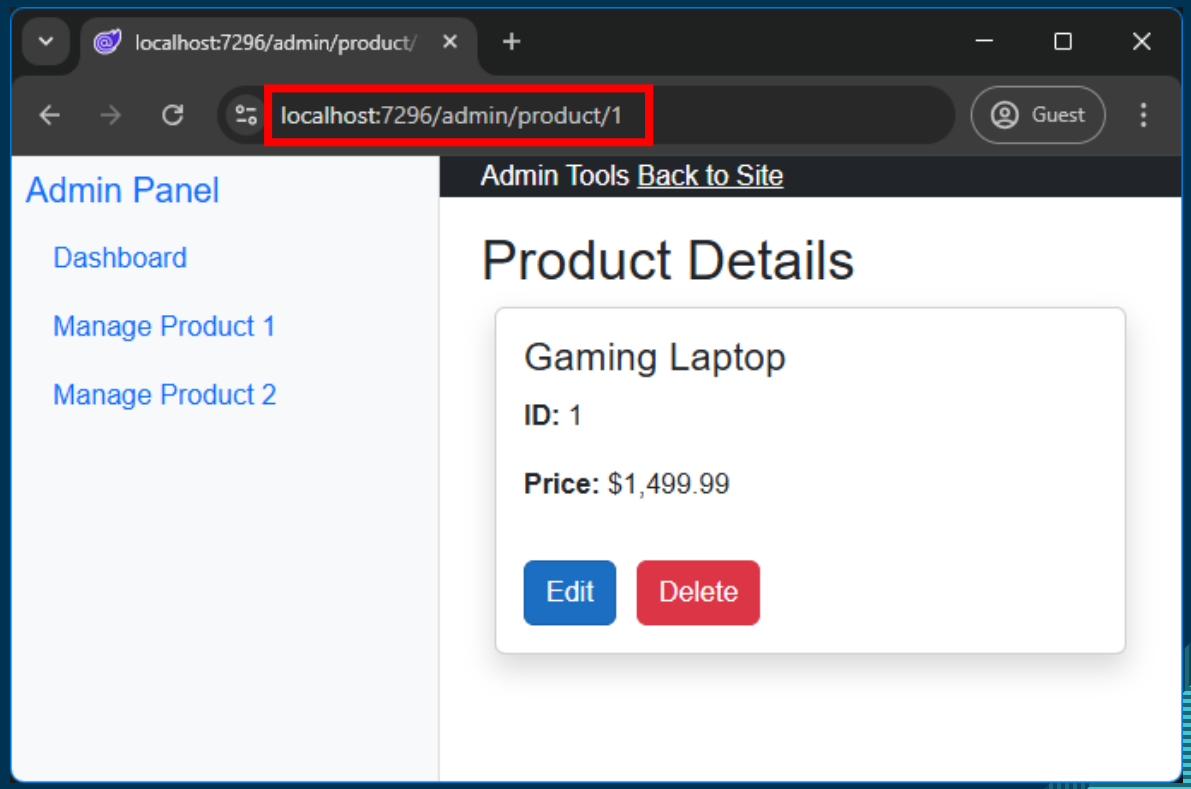
- **Improves Maintainability**:
  - Need to change a wrapper , layout style, or error UI?
  - Do it once in BaseLayout, and all nested layouts benefit.
  - Keeps each layout's job focused and clean.

- **Enforces Consistency**:
  - All pages use the same top-level structure automatically.
  - Nested layouts only handle role-specific or feature-specific layouts.

Exercise - Nested Layouts

- Use nested layouts to create separate UI for standard user and admin.
- User can only view product details whereas, admin can Edit or Delete as well.





Exercise - Nested Layouts

- Create a new Razor component file: Layout/BaseLayout.razor.
- Copy the `<div class="page">` and `<div id="blazor-error-ui">` from the `MainLayout` to `BaseLayout`.
- Add `@Body` to the `<div class="page">`.

```
@inherits LayoutComponentBase

<div class="page">
    @Body
</div>

<div id="blazor-error-ui" data-nosnippet>
    An unhandled error has occurred.
    <a href="." class="reload">Reload</a>
    <span class="dismiss">✕</span>
</div>
```

Exercise - Nested Layouts

- Go to MainLayout: Layout/MainLayout.razor.
- Change the class of `<div class="page">` to `<div class="layout-container">`.
- Delete `<div id="blazor-error-ui">`.

```
@inherits LayoutComponentBase
@layout BaseLayout

<div class="layout-container">
    <div class="sidebar">
        <NavMenu />
    </div>

Continue …
```

```
Continue …

    <main>
        <div class="top-row px-4">
            <a
href="https://learn.microsoft.com/aspnet/core/"
target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

Exercise - Nested Layouts

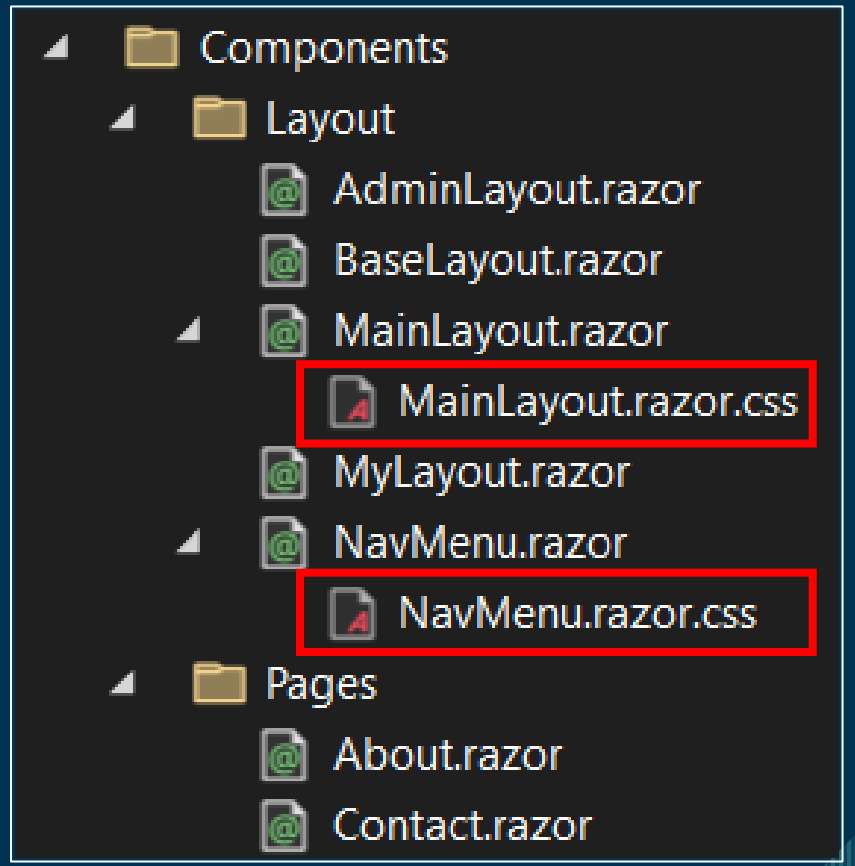
- Create a new Razor component file: Layout/AdminLayout.razor.
- You can copy-paste the code from here:

```
@layout BaseLayout
@inherits LayoutComponentBase

<div class="layout-container">
    <div class="sidebar bg-light">
        <nav class="nav flex-column p-2">
            <h5 class="text-primary">Admin Panel</h5>
            <NavLink href="/admin/dashboard"
class="nav-link">Dashboard</NavLink>
            <NavLink href="/admin/product/1"
class="nav-link">Manage Product 1</NavLink>
            <NavLink href="/admin/product/2"
class="nav-link">Manage Product 2</NavLink>
        </nav>
    </div>
Continue…
```

```
Continue…
  <main>
        <div class="top-row bg-dark text-white px-4">
            <span>Admin Tools</span>
            <a href="/" class="ms-auto text-white text-
decoration-underline">Back to Site</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

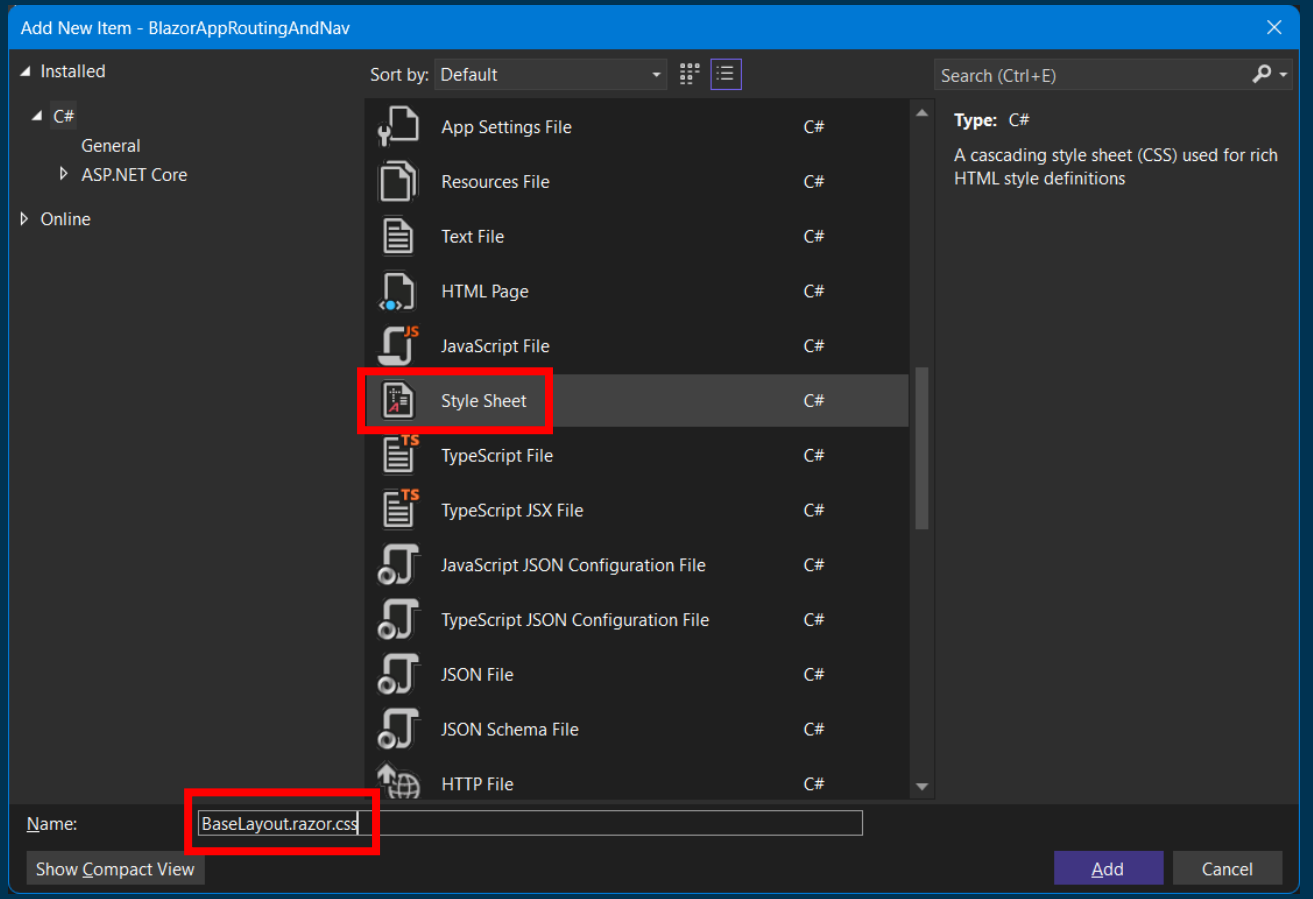
Exercise - Nested Layouts

- Now we need to fix some CSS of these Layouts.
- Notice how the built-in layouts, `MainLayout` and `NavMenu` have CSS files under their names.
- These are called Scoped CSS files.
- When you add a file named ComponentName.razor.css in the same folder as the .razor file, Blazor will:
  - Automatically scope the CSS to that component.
  - Apply styles only within that component's rendered markup.
  - Add it under the .razor file in Solution Explorer (like a linked file).



Exercise - Nested Layouts

- To create a CSS File for BaseLayout.razor, right-click on Layout folder.
- Select Add → New Item.
- Choose Style Sheet.
- Name it BaseLayout.razor.css.
- Visual Studio will automatically link it under BaseLayout.razor.



Exercise - Nested Layouts

- Once BaseLayout.razor.css created, copy-paste the CSS code from here:

```
.page {
    position: relative;
    display: flex;
    flex-direction: column;
}
#blazor-error-ui {
    color-scheme: light only;
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    box-sizing: border-box;
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}
```

```
#blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Exercise - Nested Layouts

- Go to MainLayout.razor.css.
- Delete the CSS code under `.page`.

MainLayout.razor.css | BaseLayout.razor.css

```
.page {
    position: relative;
    display: flex;
    flex-direction: column;
}

main {
    flex: 1;
}
```

- And delete `#blazor-error-ui` and `#blazor-error-ui .dismiss`.

MainLayout.razor.css | BaseLayout.razor.css | AdminLayout.razor | BaseLayo

```
#blazor-error-ui {
    color-scheme: light only;
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    box-sizing: border-box;
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

    #blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Exercise - Nested Layouts

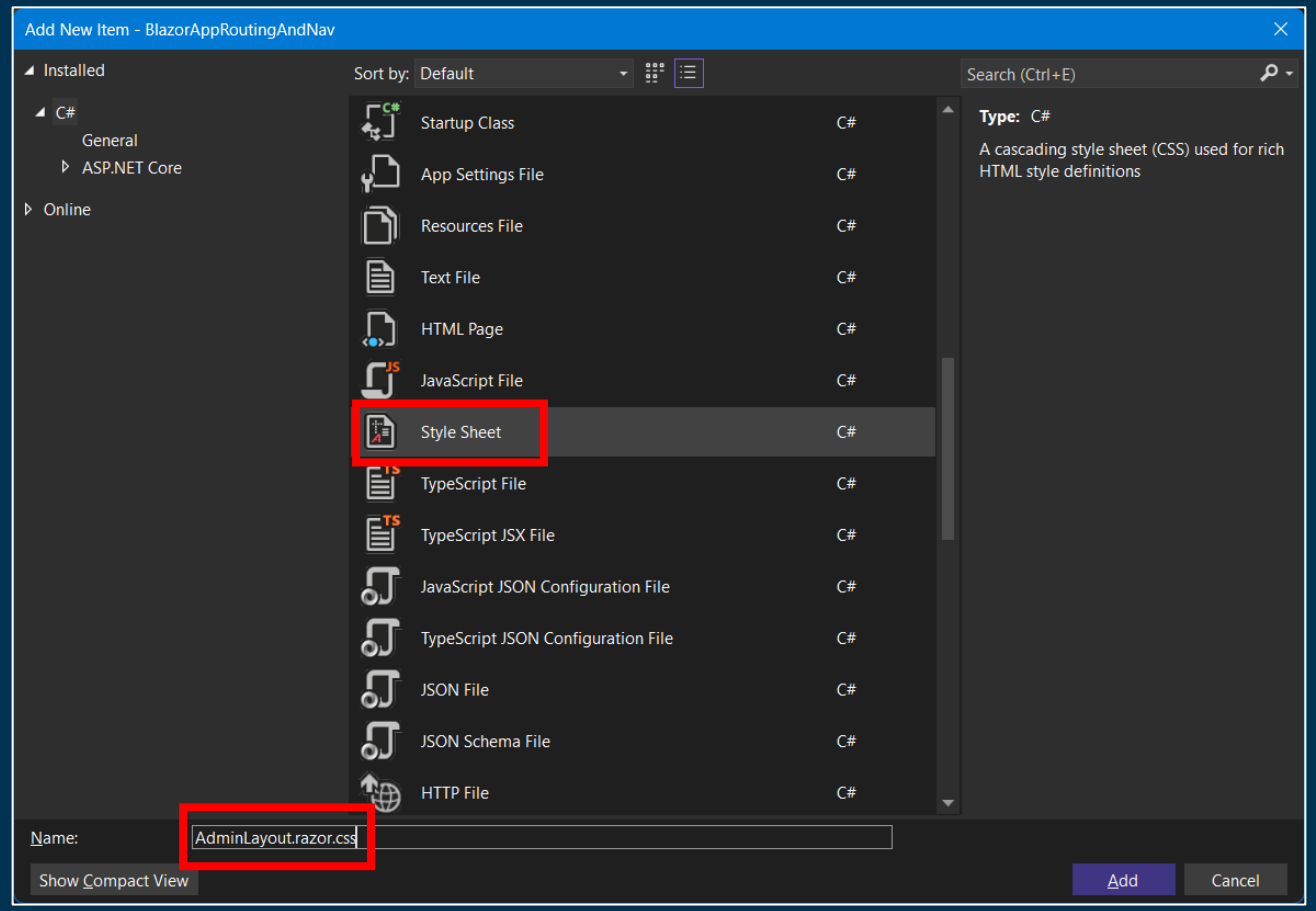
- Add CSS code for `.layout-container` to the MainLayout.razor.css.

MainLayout.razor.css | BaseLayout.razor.css

```
.layout-container {
    display: flex;
    flex: 1;
}

main {
    flex: 1;
}
```

Exercise - Nested Layouts

- Create another CSS File for AdminLayout.razor by right-clicking on Layout folder.
- Select Add → New Item.
- Choose Style Sheet.
- Name it AdminLayout.razor.css.



Exercise - Nested Layouts

- Once AdminLayout.razor.css created, copy-paste the CSS code from here:

```
.sidebar {
    width: 250px;
    height: 100vh;
    position: sticky;
    top: 0;
    border-right: 1px solid #ddd;
    background-color: #f8f9fa;
}
main {
    flex: 1;
    display: flex;
    flex-direction: column;
}
.layout-container {
    display: flex;
    flex: 1;
}
```

Exercise - Nested Layouts

- Go to Product.razor and provide the `MainLayout` as its layout.

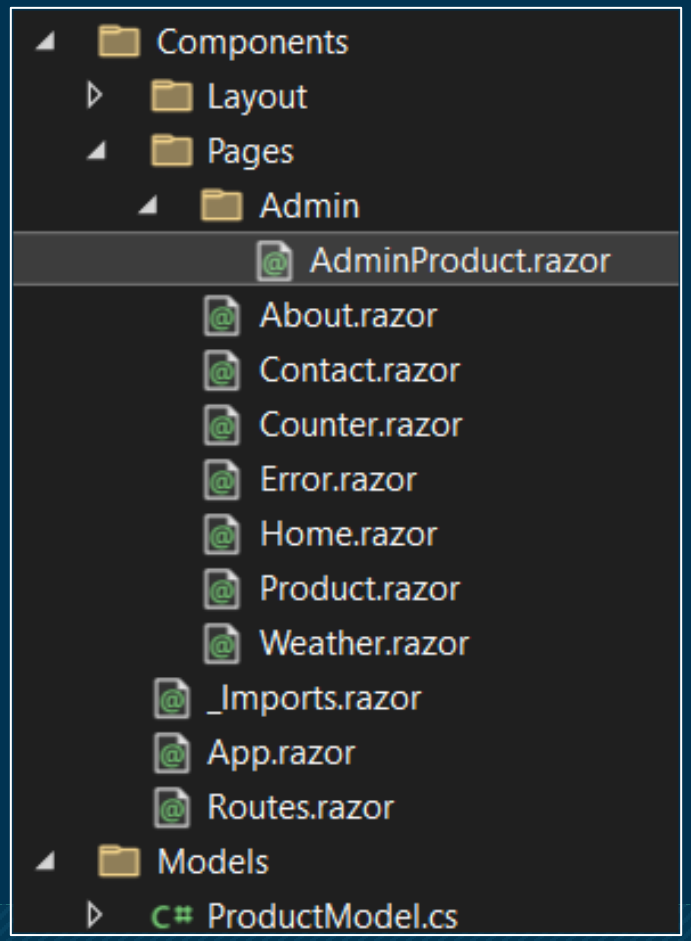
Product.razor

ndNav BlazorAppRoutingAndNav.C

```
@page "/product/{id:int}"

@layout MainLayout

<h1>Product Details</h1>

@if (product != null)
{
    <div class="card p-3 m-2 shadow">
```

Exercise - Nested Layouts

- Add a new folder Admin to the Pages folder.
- Then, add a new razor component AdminProduct.razor inside the Admin folder.



Exercise - Nested Layouts

- For AdminProduct.razor page you can copy-paste the code from here:

```
@page "/admin/product/{id:int}"
@layout AdminLayout
<h1>Product Details</h1>
@if (product != null)
{
    <div class="card p-3 m-2 shadow">
        <h4>@product.Name</h4>
        <p><strong>ID:</strong> @product.Id</p>
        <p><strong>Price:</strong>
@product.Price.ToString("C")</p>
        <!-- Dummy admin actions -->
        <div class="mt-3">
            <button class="btn btn-
primary">Edit</button>
            <button class="btn btn-danger ms-
2">Delete</button>
        </div>
    </div>
}
```

```
else
{
    <div class="alert alert-warning">
        Product not found. <NavLink href="/">Go back
to home</NavLink>
    </div>
}

Continue …
```

Exercise - Nested Layouts

- For AdminProduct.razor page you can copy-paste the code from here:

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

    private ProductModel? product;

    private List<ProductModel> products = new()
    {
        new ProductModel(1, "Gaming Laptop",
1499.99M),
        new ProductModel(2, "Wireless Mouse",
29.99M),
        new ProductModel(3, "Mechanical Keyboard",
79.50M),
        new ProductModel(4, "27-inch Monitor",
299.00M),
        new ProductModel(5, "USB-C Dock", 59.95M)
    };
Continue …
```

```
Continue …

    protected override void OnInitialized()
    {
        product = products.Find(p => p.Id == Id);
    }
}
```

Exercise - Nested Layouts

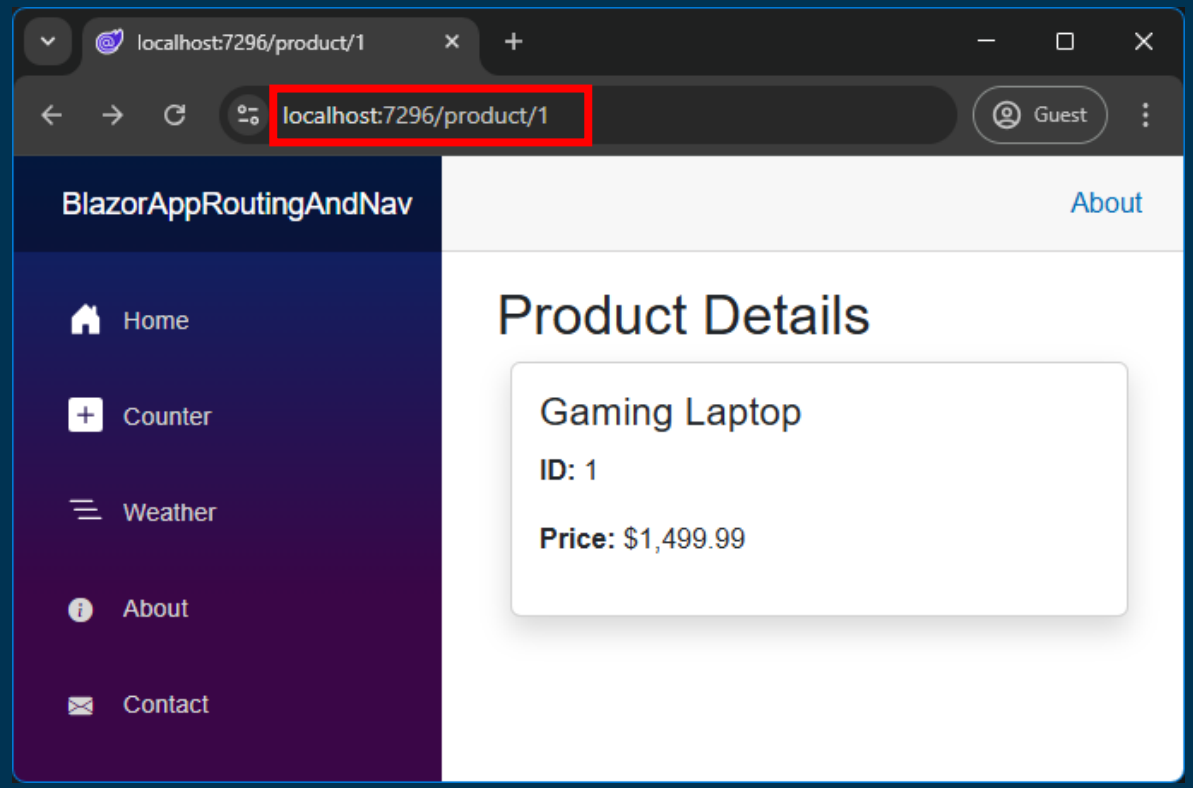
- The `@page` directive points to the route `/admin/product/{id}`.
- It uses `AdminLayout` as its layout.
- There are some dummy buttons to simulate Edit and Delete functionality.
  - *The buttons don't work in this example.*
- The remainder code is like the one in Product.razor component.

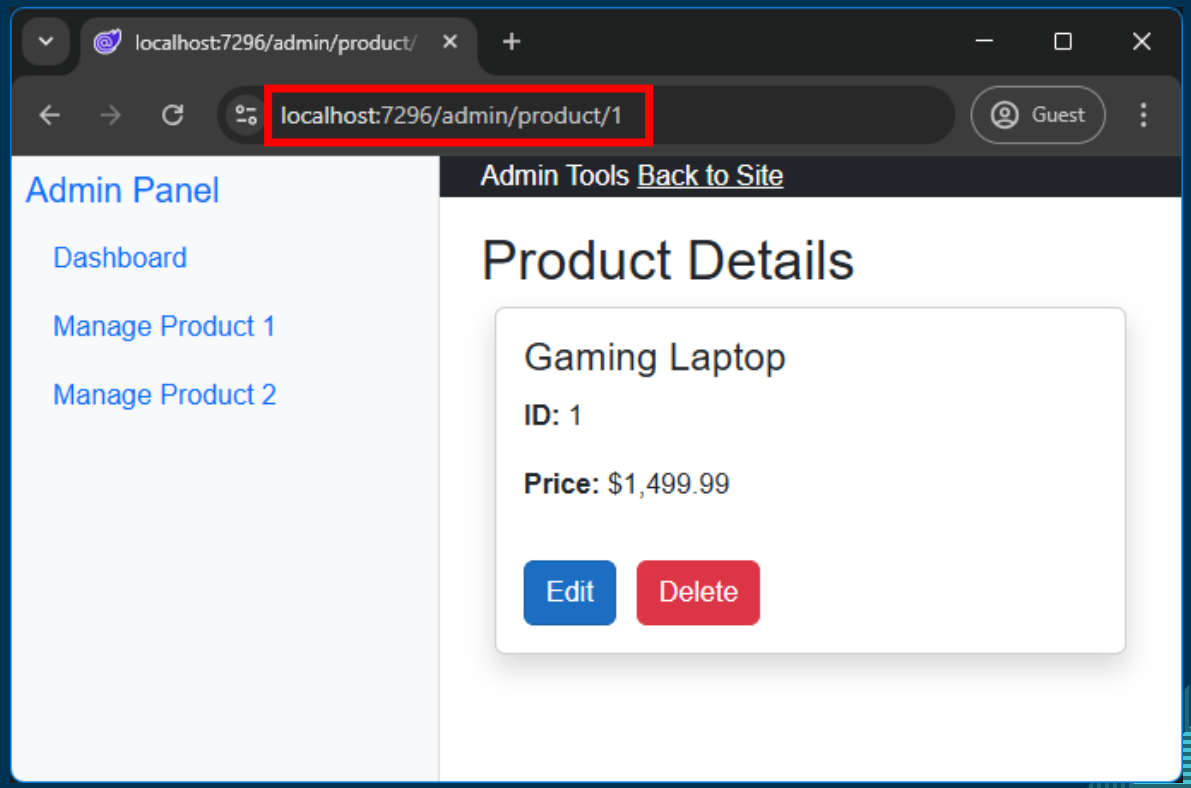
AdminProduct.razor | Product.razor

```
@page "/admin/product/{id:int}"

@layout AdminLayout

<h1>Product Details</h1>

@if (product != null)
{
    <div class="card p-3 m-2 shadow">
        <h4>@product.Name</h4>
        <p><strong>ID:</strong> @product.Id</p>
        <p><strong>Price:</strong> @product.Price.ToString("C")</p>

        <!-- Dummy admin actions -->
        <div class="mt-3">
            <button class="btn btn-primary">Edit</button>
            <button class="btn btn-danger ms-2">Delete</button>
        </div>
    </div>
}
```

Exercise - Nested Layouts

- Run the app and try navigating to:
  - `/product/1`
  - `/admin/product/1`





Do It Yourself!

- **Show User Profile by ID**:
  - Create a new Razor component: UserProfile.razor.
  - Accept a route parameter for `id`:

```
@page "/user/{id:int}"
```

  - Display a fake user profile based on the `id`.
  - Use a list like:

```
private List<User> Users = new()
{
    new User { Id = 1, Name = "Alice" },
    new User { Id = 2, Name = "Bob" }
};
```

  - Test with `/user/1`, `/user/2`, etc.

Do It Yourself!

- **Optional Route Parameter with Fallback**:
  - Create a page: Welcome.razor.
  - Define a route like:

```
@page "/welcome"
@page "/welcome/{name}"
```

  - If name is supplied, show “Welcome, [name]”.
  - If not, show a default message: “Welcome, Guest”.

Do It Yourself!

- **Admin vs. User Layout (UI Only)**:
  - Create two layouts:
    - UserLayout.razor: Regular sidebar and top bar.
    - AdminLayout.razor: Sidebar includes links to “Dashboard”, “Manage Users”, etc.
  - Create two pages:
    - UserHome.razor using UserLayout.
    - AdminHome.razor using AdminLayout.
  - Add fake links like:
    - `/user/home` → standard view.
    - `/admin/home` → admin view.

Thank You

References

Some material has been taken from:

- Use pages, routing, and layouts to improve Blazor navigation:
  - https://learn.microsoft.com/en-us/training/modules/use-pages-routing-layouts-control-blazor-navigation/1-introduction

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.