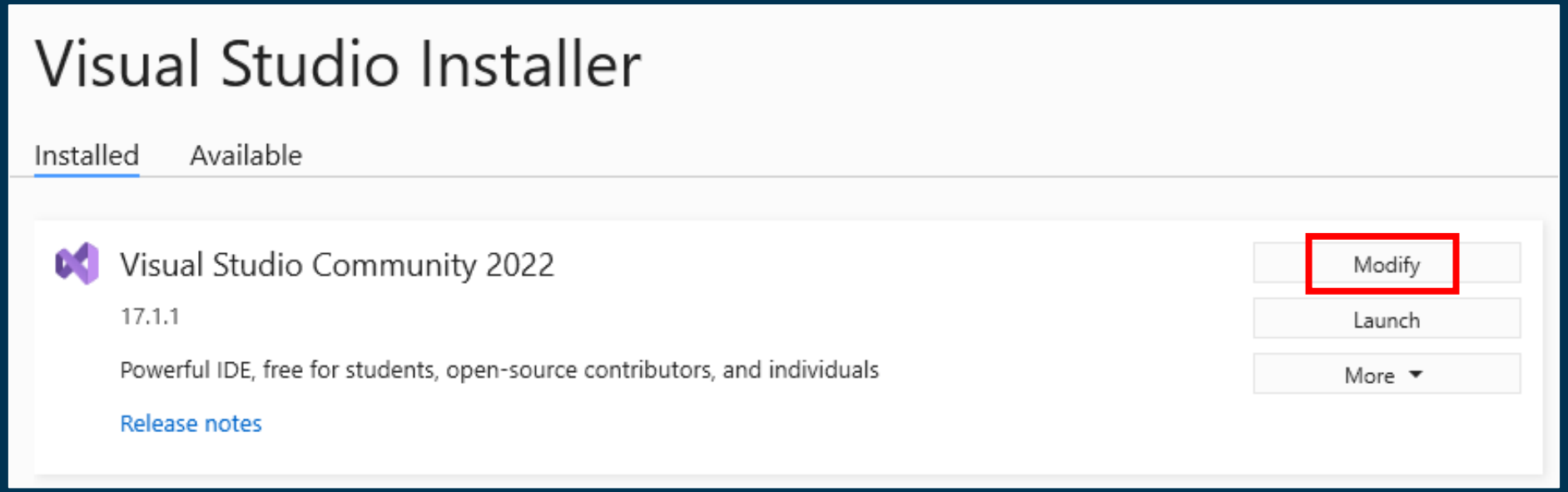
Introduction to ASP.NET Blazor

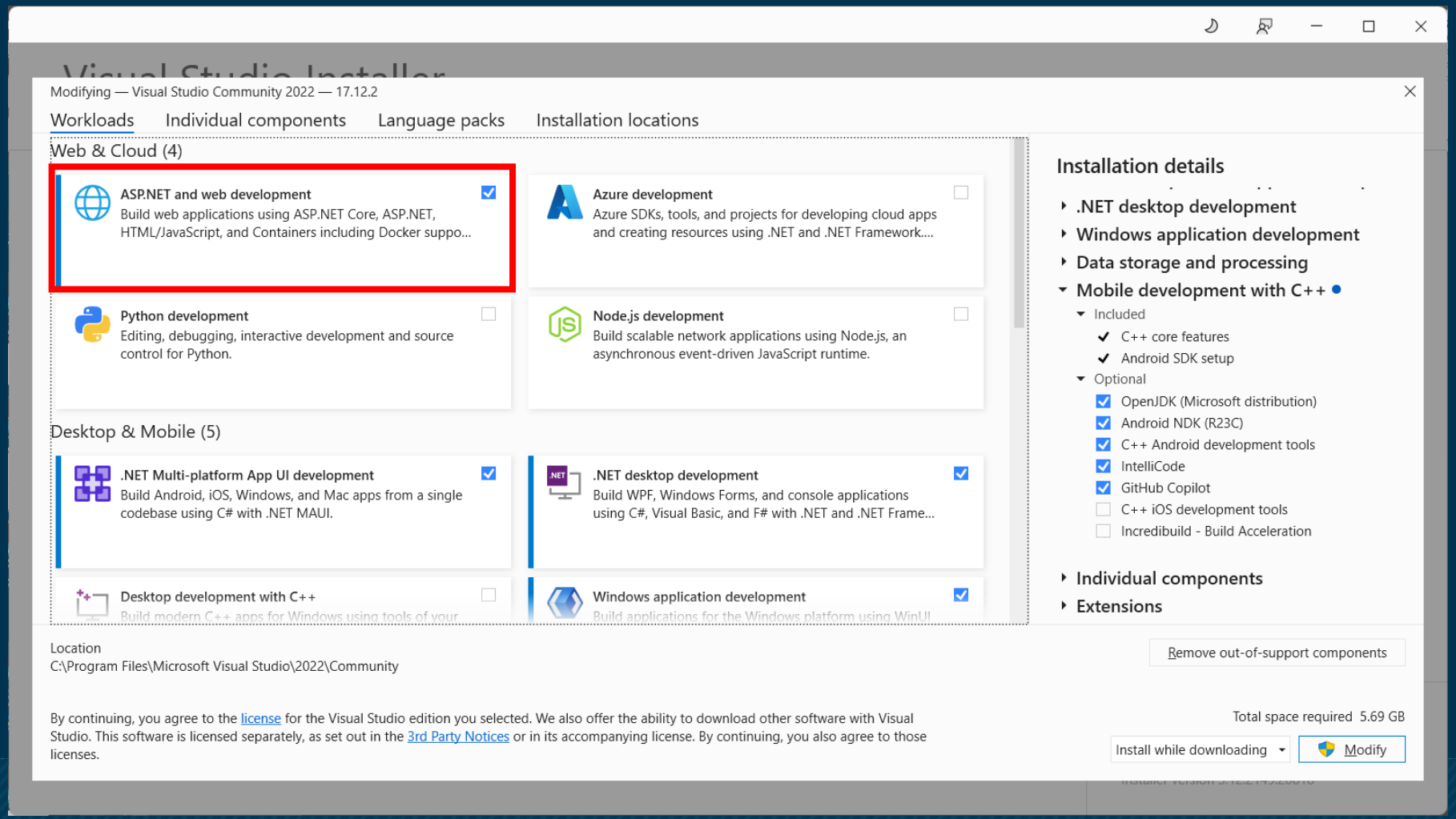
Installing ASP.NET and Web Development Workload in Visual Studio

- You may not have the ASP.NET and Web Development workload installed in Visual Studio.
- To install, you must run the Visual Studio Installer again.
- You can do this from the Start menu by searching for Visual Studio Installer.
- When you are presented with the list of installed products, click Modify.



Installing ASP.NET and Web Development Workload in Visual Studio

- Then check the ASP.NET and Web Development option before clicking Modify to install the selected options.



Blazor Web Apps in VS Code

- Blazor Web Apps can be developed in VS Code as well.
- VS Code instructions can be found here:
  - https://dotnet.microsoft.com/en-us/learn/aspnet/blazor-tutorial/intro

Introduction to Web Apps

- A web app (short for web application) is a software program that runs in a web browser and is accessed through the internet or an intranet.
- Unlike traditional desktop applications, you don't need to install anything to use a web app.
  - Just a browser like Chrome, Firefox, Safari, or Edge.

- Key Features of a Web App:
  - Runs in a browser: You access it through a URL (e.g., https://www.google.com).
  - No installation required: Everything is stored and runs on a server.
  - Cross-platform: Works on any device with a browser (PC, tablet, smartphone).
  - Interactive: Unlike regular websites, web apps allow user input and interaction.

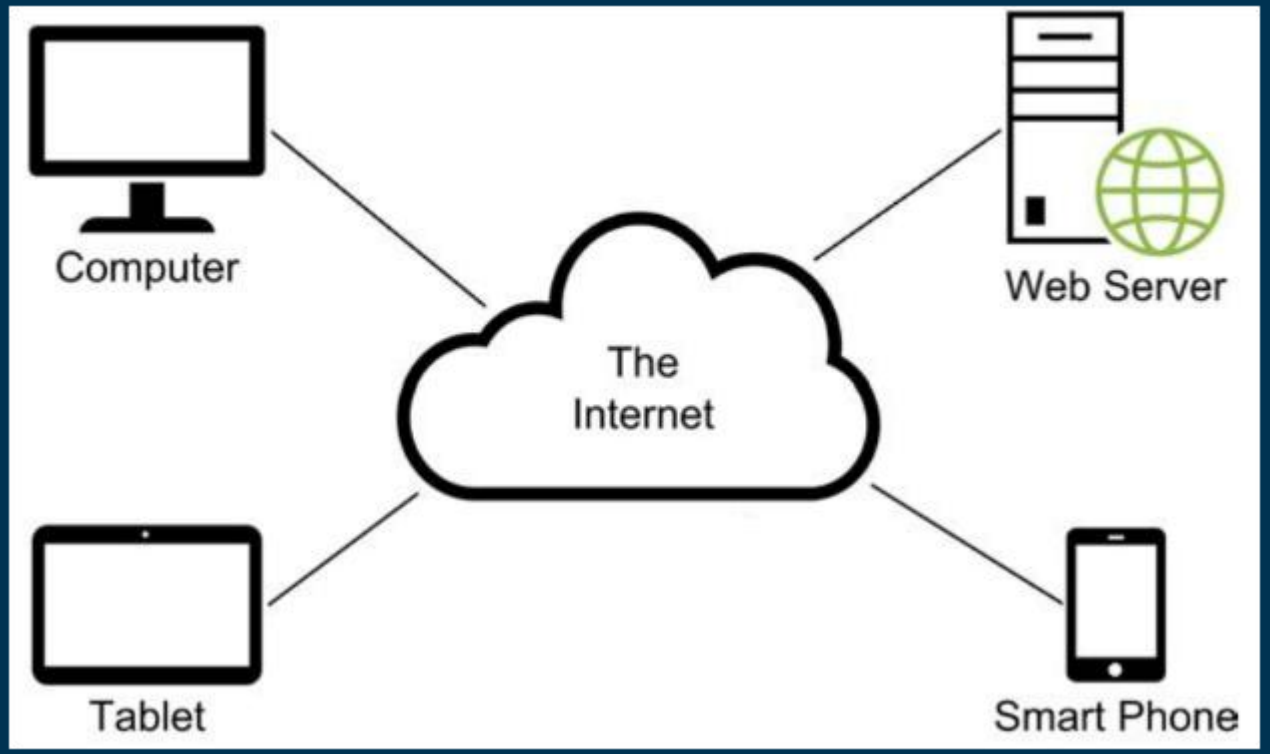
Introduction to Web Apps

- Examples of Web Apps:
  - Gmail: email client
  - Google Docs: online word processor
  - Facebook: social media platform
  - Spotify Web Player: music streaming

- Web App vs Website:
  - A website is mostly informational (like blogs or company sites).
  - A web app is dynamic and interactive (you can log in, fill out forms, manage data, etc.).

Introduction to Web Apps

- A web app consists of clients, a web server, and a network.
- The clients use web browsers to request web pages from the web server.
- Clients are often computers, smart phones, or tablets.
- The web server returns the pages, that are requested, to the browser.
- A network connects the clients to the web server.

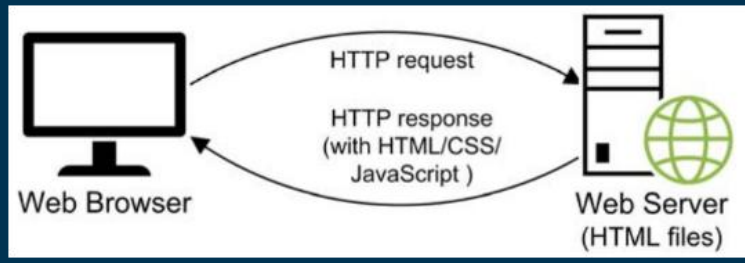


Static Web Page

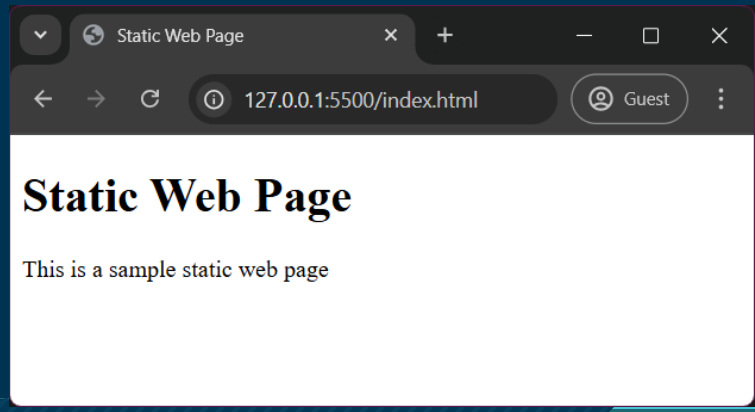
- A static web page is a simple web page that displays same content to every visitor.
- It is written in plain HTML, CSS, JS and does not involve any server-side scripting.

- Fixed Content: The content doesn't change unless someone manually edits the HTML file.
- Fast Loading: Since it doesn't rely on databases or server-side code, it loads quickly.
- Simple to Create: Just need HTML, CSS and JS.
- No User Interaction: It doesn't respond to user inputs (like logins or form submissions) in real-time.

- **Examples**:
  - Personal portfolios.
  - Company "About Us" pages.
  - Simple informational websites (e.g., small business brochures).

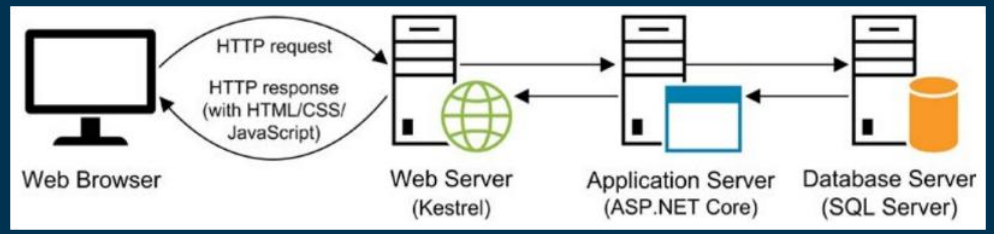



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Static Web Page</title>
    </head>

    <body>
        <h1>Static Web Page</h1>
        <p>This is a sample static web page</p>
    </body>
</html>
```



Dynamic Web Page

- A dynamic web page displays different content and allows interaction based on user input, time, database content, or other variables.
- Unlike static pages, dynamic pages are generated in real-time using server-side technologies like ASP.NET, PHP, Node.js, or frameworks like React, Angular, etc.



- Content Changes Automatically: Can show different information to different users (e.g., news feeds, weather updates, user dashboards).
- Database Driven: Often pulls data from a database (e.g., SQL Server, MySQL).
- Interactive: Supports user interaction like logging in, posting comments, searching, or filtering content.
- Backend Logic: Has scripts running on the server to handle data processing and user input.

- **Examples**:
  - Social media platforms (e.g., Facebook, Instagram).
  - E-commerce sites (e.g., Amazon, eBay).
  - Online banking portals.
  - News websites that update frequently.

Introduction to ASP.NET Core

- ASP.NET Core is a modern, open-source, cross-platform framework developed by Microsoft for building web applications, APIs, and microservices.
- It's a rebuild of the original ASP.NET framework, and is designed for performance, flexibility, and scalability.

- ASP.NET Core can be used for:
  - Building websites.
  - Creating RESTful APIs.
  - Developing real-time apps (like chat apps).
  - Hosting microservices and cloud-native apps.

Members of the ASP.NET Core Family

- **Blazor**:
  - Build interactive web UIs with C# (instead of JavaScript or other tools).
  - This module is based on ASP.NET Blazor (https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor)
- **ASP.NET Core MVC**:
  - Model-View-Controller pattern for building enterprise web apps and APIs.
  - Learn more: https://dotnet.microsoft.com/en-us/apps/aspnet/mvc
- **Razor Pages**:
  - Simpler page-focused approach to web UI.
  - Learn more: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/
- **SignalR**:
  - Real-time web functionality (e.g., chat, live dashboards).
  - Learn more: https://dotnet.microsoft.com/en-us/apps/aspnet/signalr
- **Minimal APIs**:
  - Lightweight way to build REST APIs with minimal boilerplate code.
  - Learn more: https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api/

Introduction to ASP.NET Blazor

- Blazor is a web framework from Microsoft that lets you build interactive web applications.
- It is a modern frontend web framework based on HTML, CSS, and C#.
- It's part of the ASP.NET Core family and is great for developers who are already familiar with .NET and want to build rich web UIs without switching to other frameworks.

- Key Features of Blazor:
  - C# Everywhere: Handle UI events from the browser and implement logic all in C#.
  - Component-Based: Build reusable UI components (like in React or Angular).
  - One Development Stack: Build entire web app from the frontend to the backend using a single development stack.
  - WebAssembly Support: Runs in the browser using WebAssembly (https://webassembly.org/).
  - Server Hosting: Runs on the server (Blazor Server).
  - Full .NET Integration: Access .NET libraries, tools, and packages.

Types of Blazor

Blazor WebAssembly (WASM):

- Runs completely in the browser.
- Downloads the .NET runtime to the browser.
- Works offline once loaded.
- Ideal for static hosting (e.g., GitHub Pages, Azure Static Web Apps).

Blazor Server:

- Executes code on the server.
- Uses SignalR to update the UI via a persistent connection.
- Smaller initial load, good for enterprise apps.
- Requires constant internet connection.

Blazor Components

- Blazor apps are built from components.
- A Blazor component is a reusable piece of web UI.
- A Blazor component encapsulates both its rendering and UI event handling logic.
- Blazor includes various built-in components for form handling, user input validation, displaying large data sets, authentication, and authorization.
- Developers can also build and share their own custom components, and many prebuilt Blazor components are available from the Blazor ecosystem.

Blazor Uses Standard Web Technologies

- You create Blazor components using Razor syntax, a convenient mixture of HTML, CSS, and C#.
- A Razor file contains plain HTML and then C# to define any rendering logic, like for conditionals, control flow, and expression evaluation.
- Razor files are then compiled into C# classes that encapsulate the component's rendering logic.

UI Event Handling and Data Binding

- Interactive Blazor components can handle standard web UI interactions using C# event handlers.
- Components can update their state in response to UI events and adjust their rendering accordingly.
- Blazor also includes support for two-way data binding to UI elements as a way to keep component state in sync with UI elements.

Blazor Example

- This example is a simple Blazor counter component implemented in Razor.

- Most of the content is HTML, while the `@code` block contains C#.
- Every time the button is pressed the `IncrementCount` method is invoked, which increments the `currentCount` field, and then the component renders the updated value.

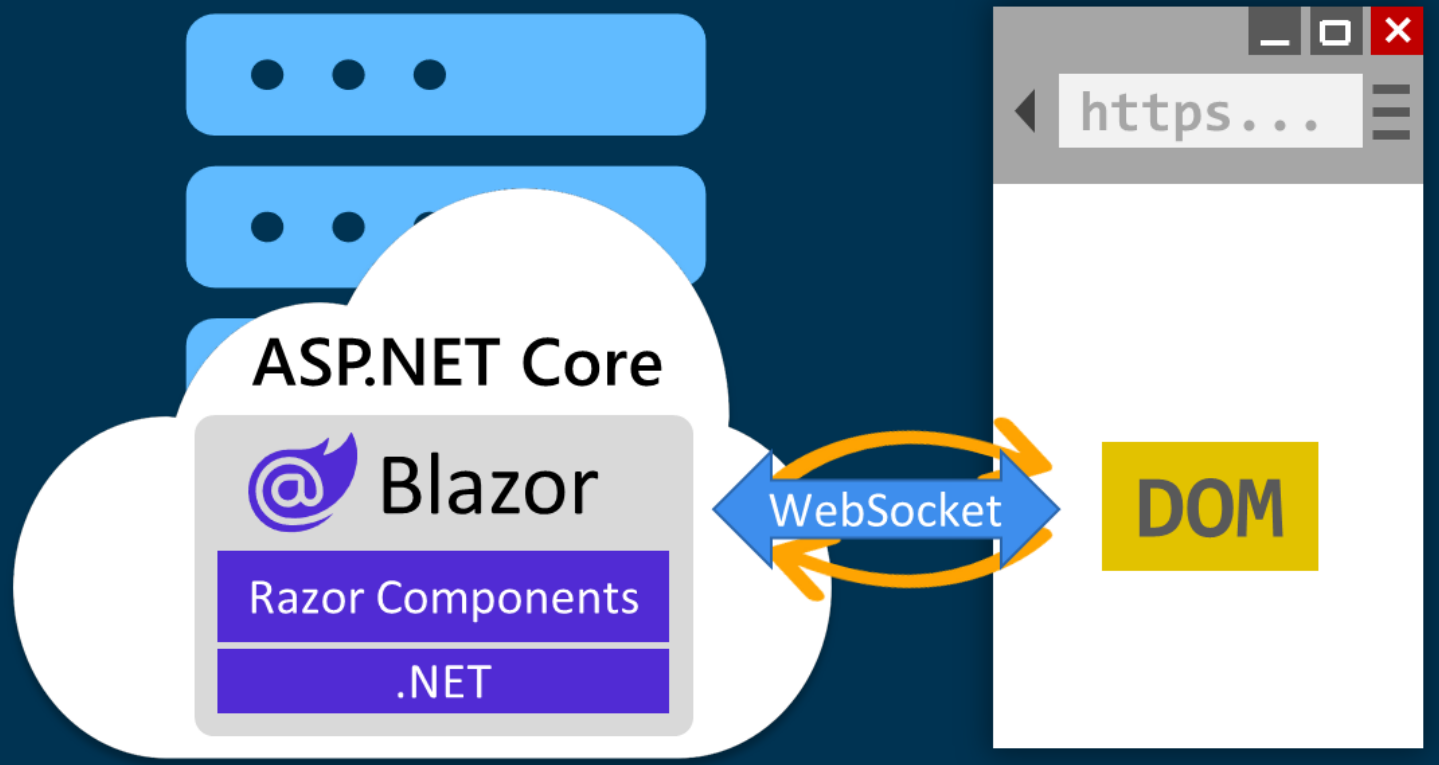
```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Server and Client-Side Rendering

- Blazor supports both server and client-side rendering of components to handle various web UI architectures.
- Components rendered from the server can access server resources, like databases and backend services.
- By default, Blazor components are rendered from the server, generating HTML in response to requests.

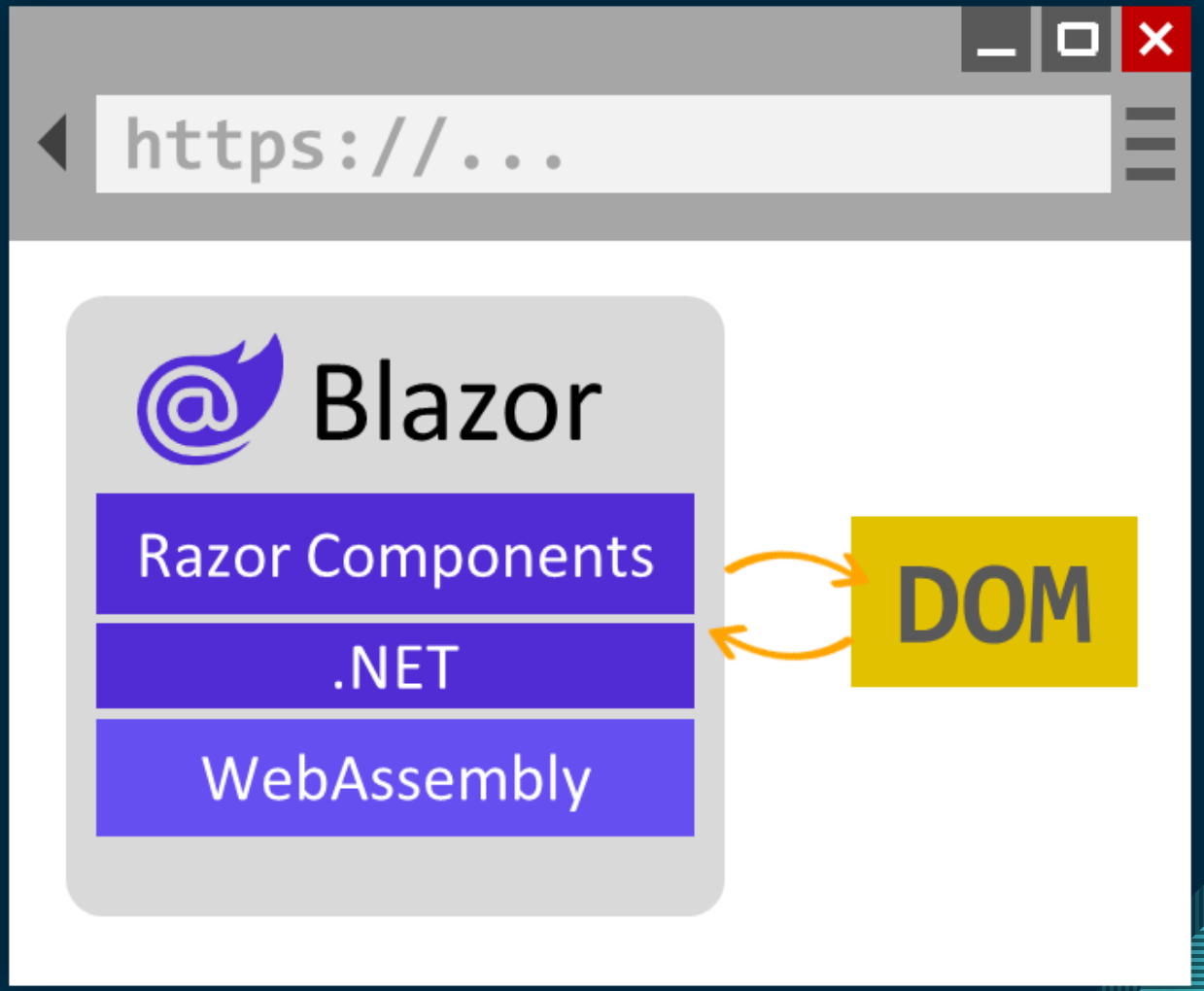
Server and Client-Side Rendering

- You can also configure server components to be interactive, so they can handle UI events, maintain state across interactions, and render updates dynamically.
- Interactive server components handle UI interactions and updates over a WebSocket connection with the browser.



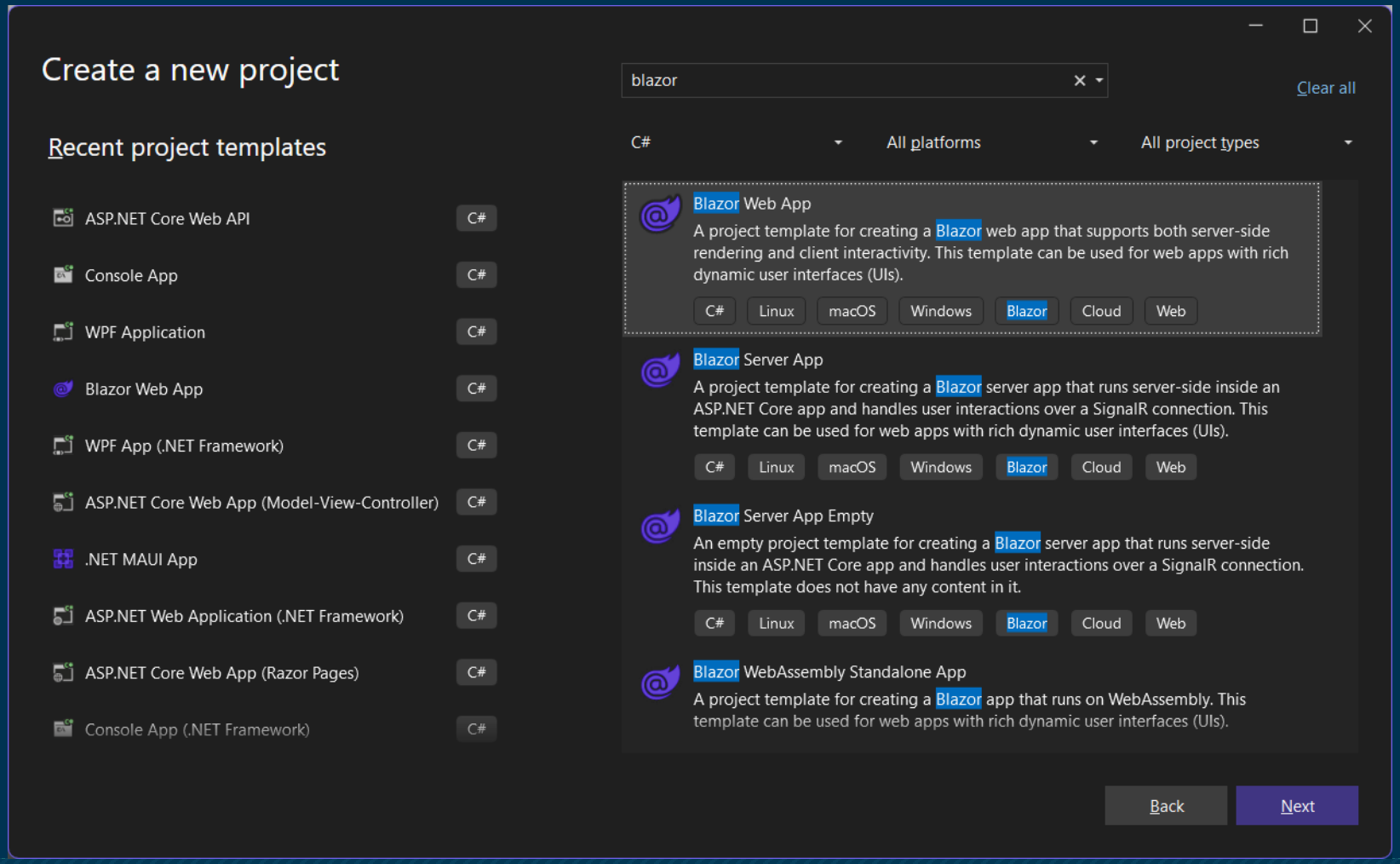
Server and Client-Side Rendering

- Alternatively, Blazor components can be rendered interactively from the client.
- The component is downloaded to the client and run from the browser via WebAssembly.
- Interactive WebAssembly components can access client resources through the web platform, like local storage and hardware, and can even function offline once downloaded.



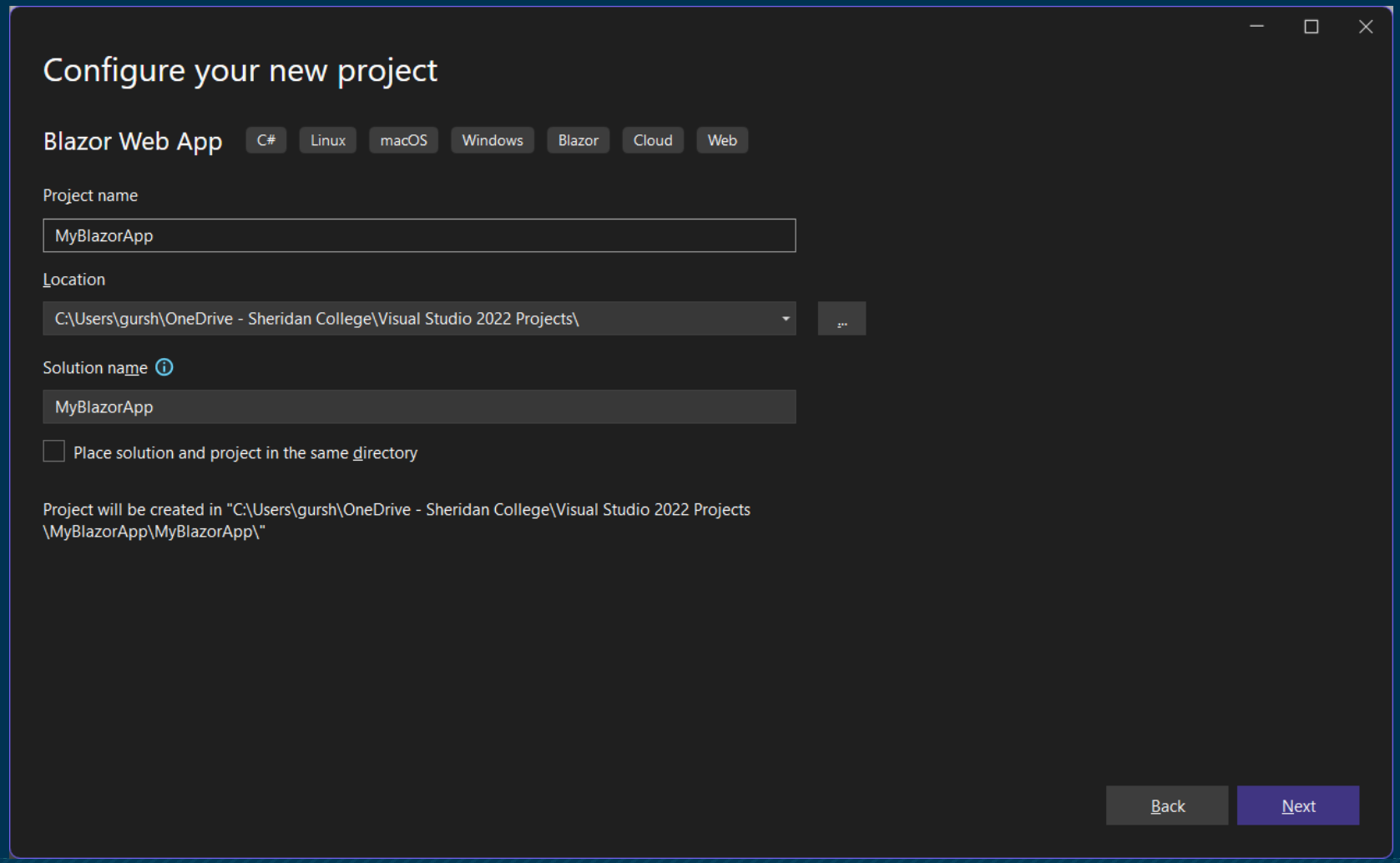
Create a Blazor Web App

- Create a new Blazor Web App.



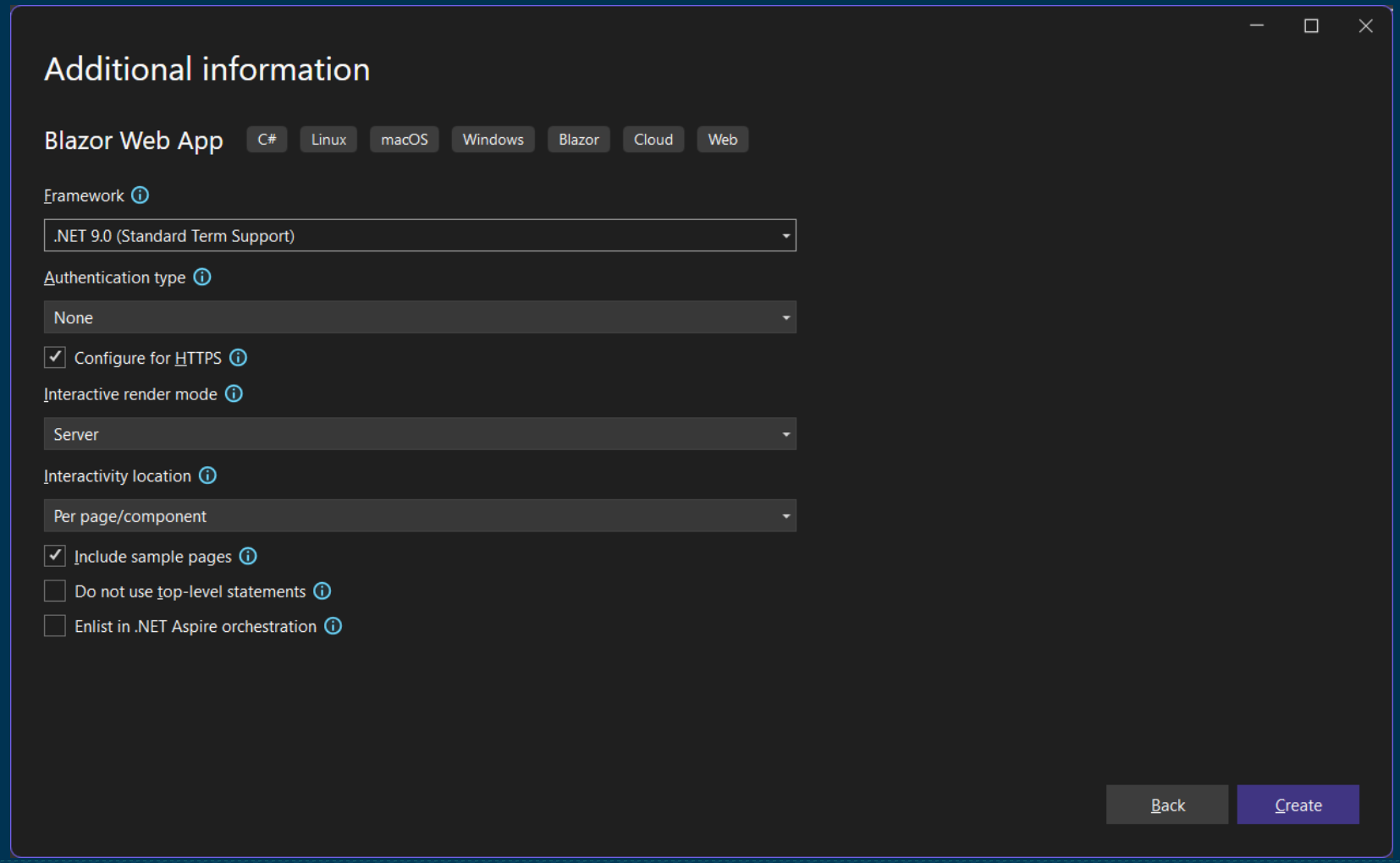
Create a Blazor Web App

- Give it a name and location.



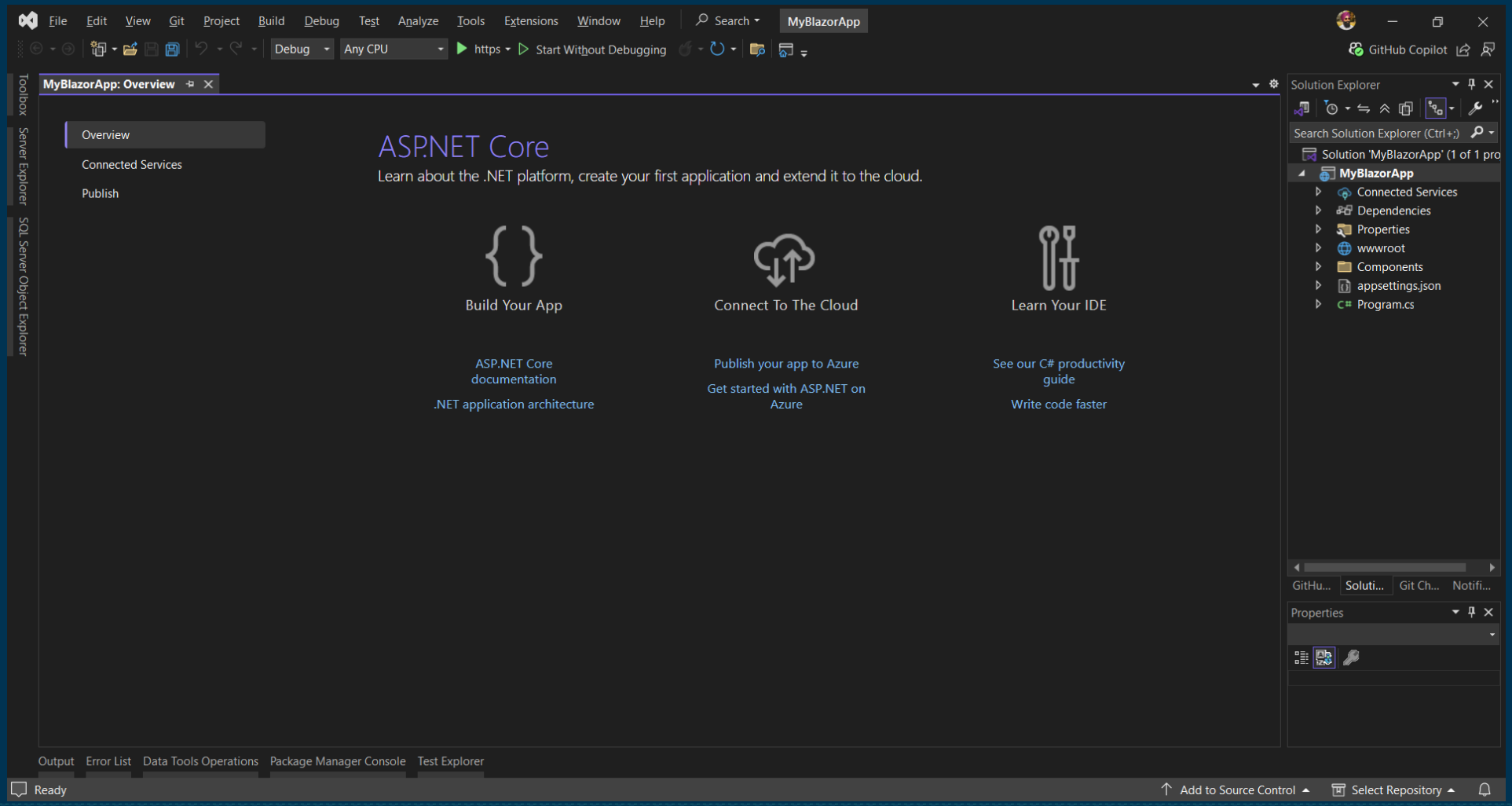
Create a Blazor Web App

- Leave these as defaults and click Create.



Create a Blazor Web App

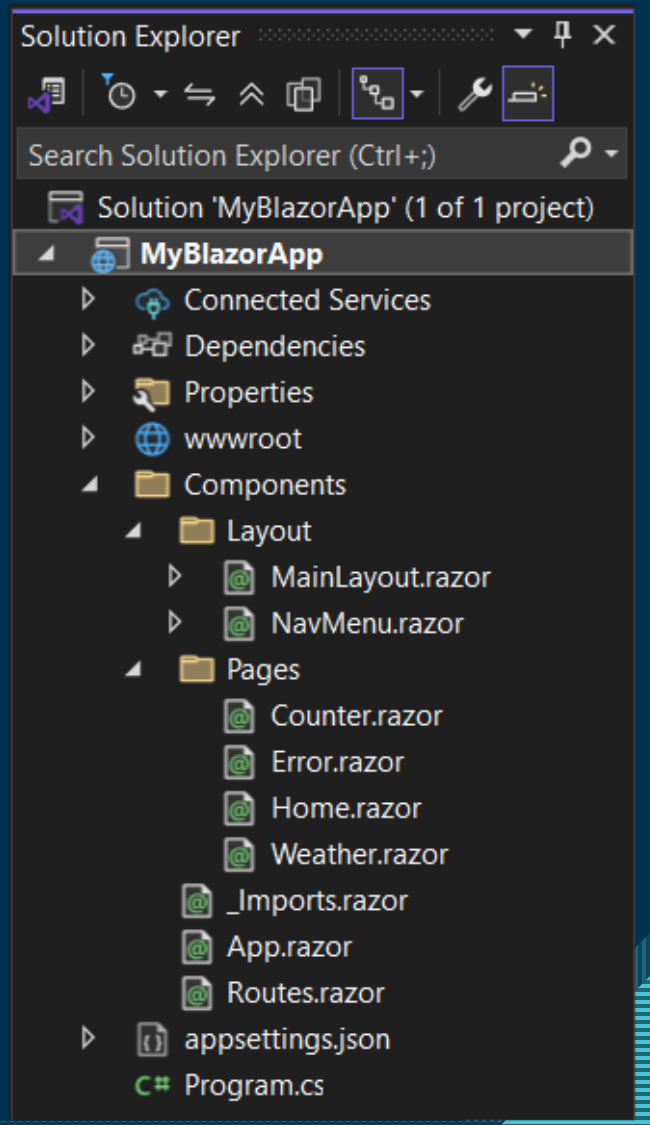
- Your project is created and loaded in Visual Studio.



Blazor Web App File Structure

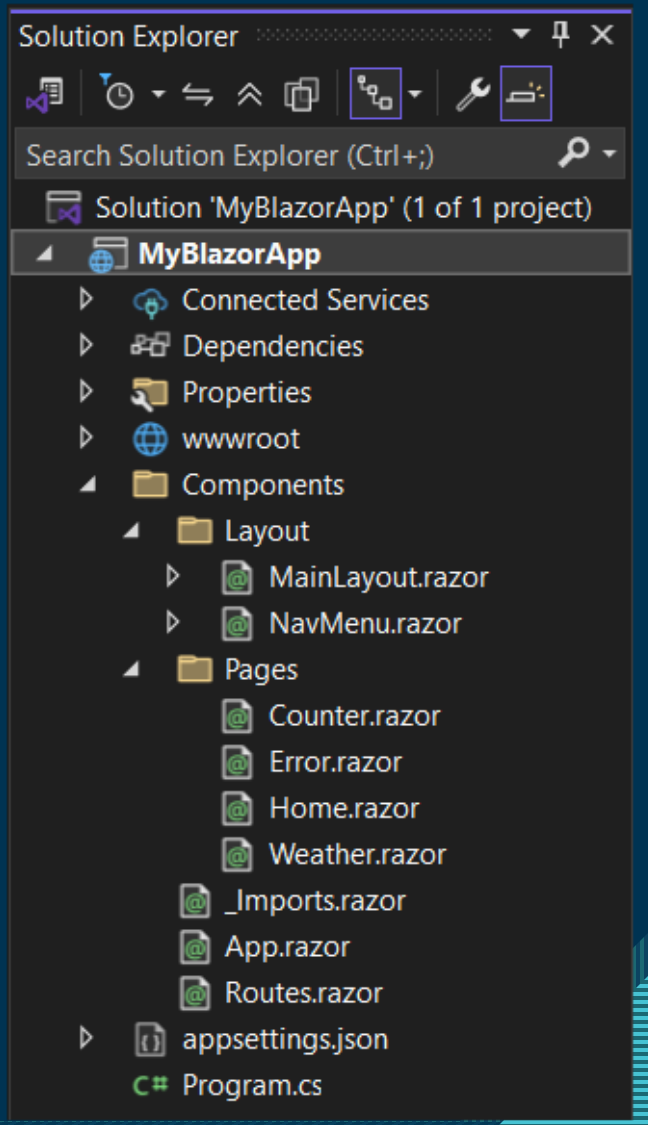
- Take a look at the contents of your project using Solution Explorer.

- Connected Services:
  - Used to connect your app to external services like Azure, REST APIs, or databases.
  - Often empty unless you configure something specific.
- Dependencies:
  - Holds all NuGet packages your project relies on, including .NET libraries, third-party packages, etc.
- Properties:
  - Contains project-level metadata such as launchSettings.json, which defines how your app runs during development (e.g., port number, environment variables).
- wwwroot:
  - The public folder, similar to the www or static directory in other frameworks.
  - Used for:
    - Static files like images, CSS, JavaScript
    - favicon.ico, logos
    - Client-side libraries



Blazor Web App File Structure

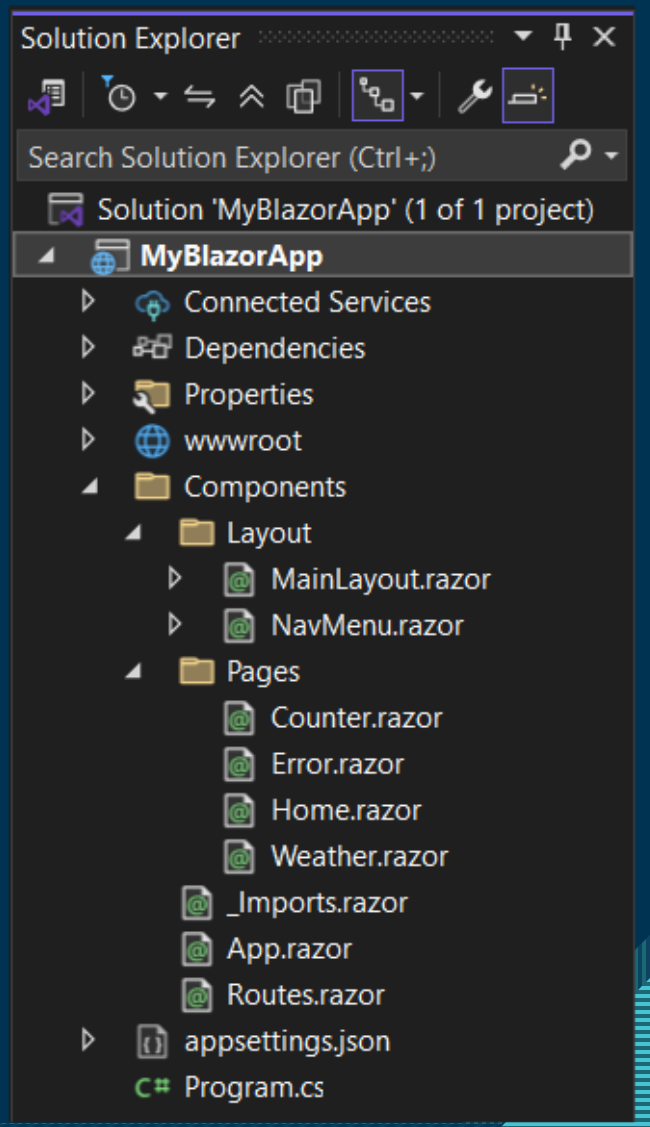
- Components:
  - Holds reusable UI building blocks (also called Razor components).
- Layout:
  - MainLayout.razor: Defines the general page structure (e.g., header, sidebar, footer).
  - NavMenu.razor: The navigation menu component usually displayed in the sidebar.
  - These layout files act like templates shared across multiple pages.
- Pages:
  - Holds the actual web pages users can navigate to. Each .razor file is a page or view.
  - Counter.razor: A sample page that demonstrates interactivity (click button to increase count).
  - Error.razor: Error-handling page.
  - Home.razor: The default landing page.
  - Weather.razor: Typically shows a weather forecast sample.
  - _Imports.razor: A special file for global `@using` statements, shared across components in the folder.
  - App.razor: The root component of your app. This sets up routing and includes layouts.
  - Routes.razor: Manages routing logic.



Blazor Web App File Structure

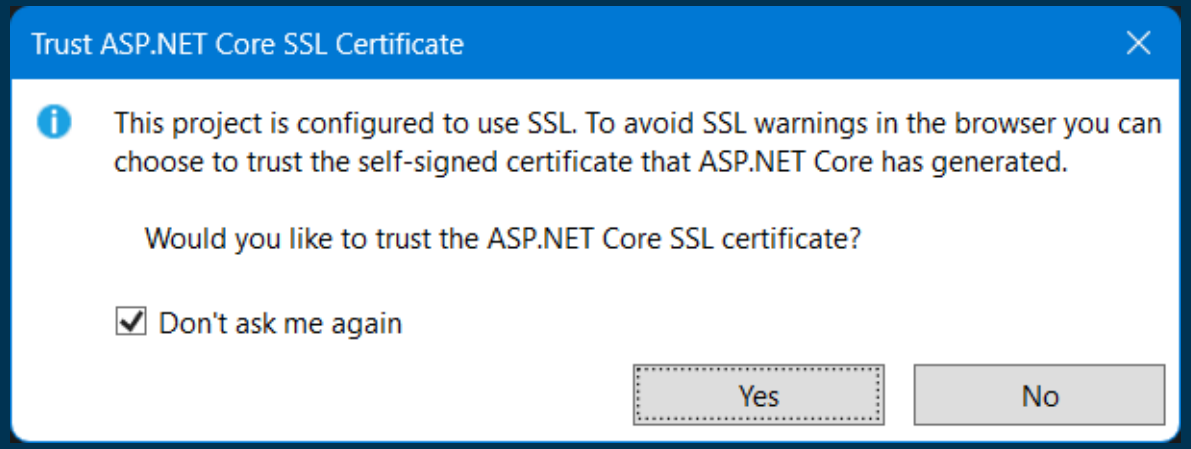
- appsettings.json:
  - Used for configuration settings like endpoints, connection strings, etc.
  - Similar to web.config in older .NET apps.
- Program.cs:
  - The entry point of your Blazor app.
  - It configures services and starts the application.

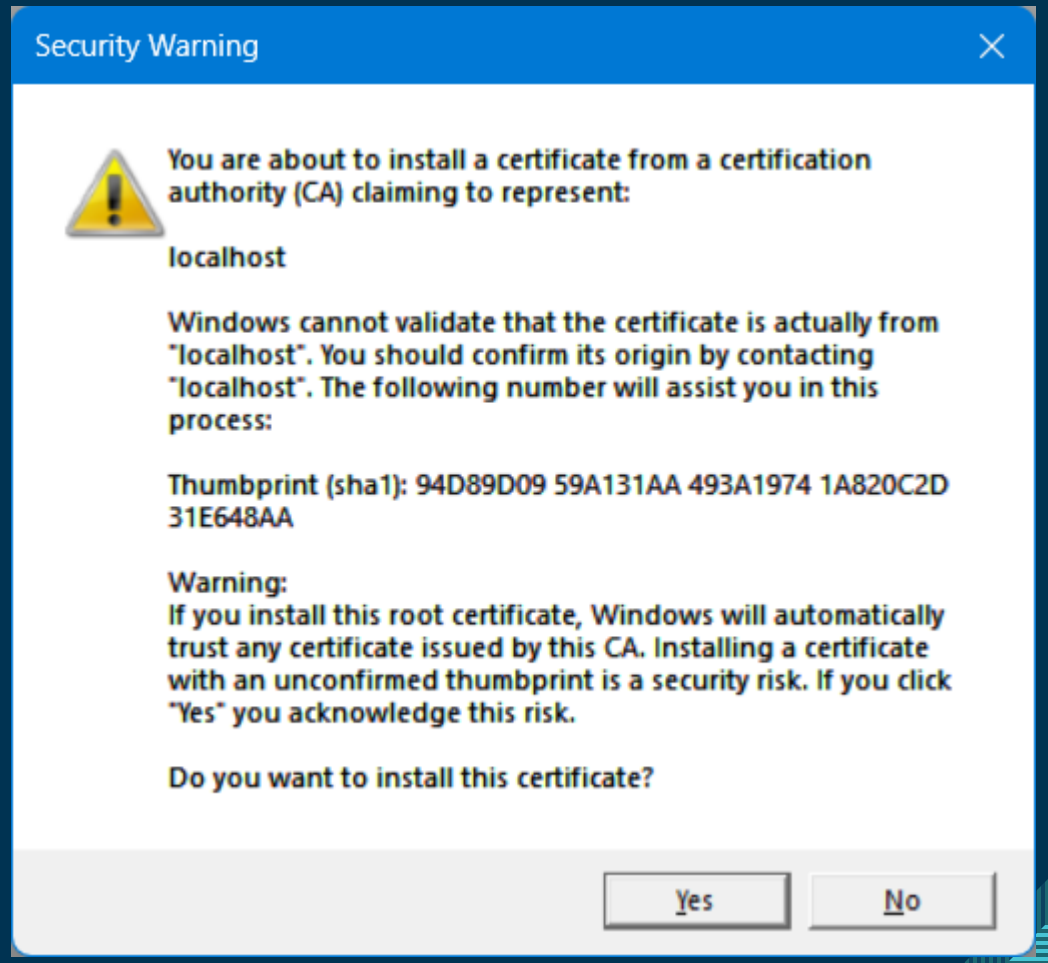
```
using MyBlazorApp.Components;

var builder = WebApplication.CreateBuilder(args);

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

var app = builder.Build();

app.UseHttpsRedirection();
app.UseAntiforgery();

app.MapStaticAssets();
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.Run();
```



Run the App

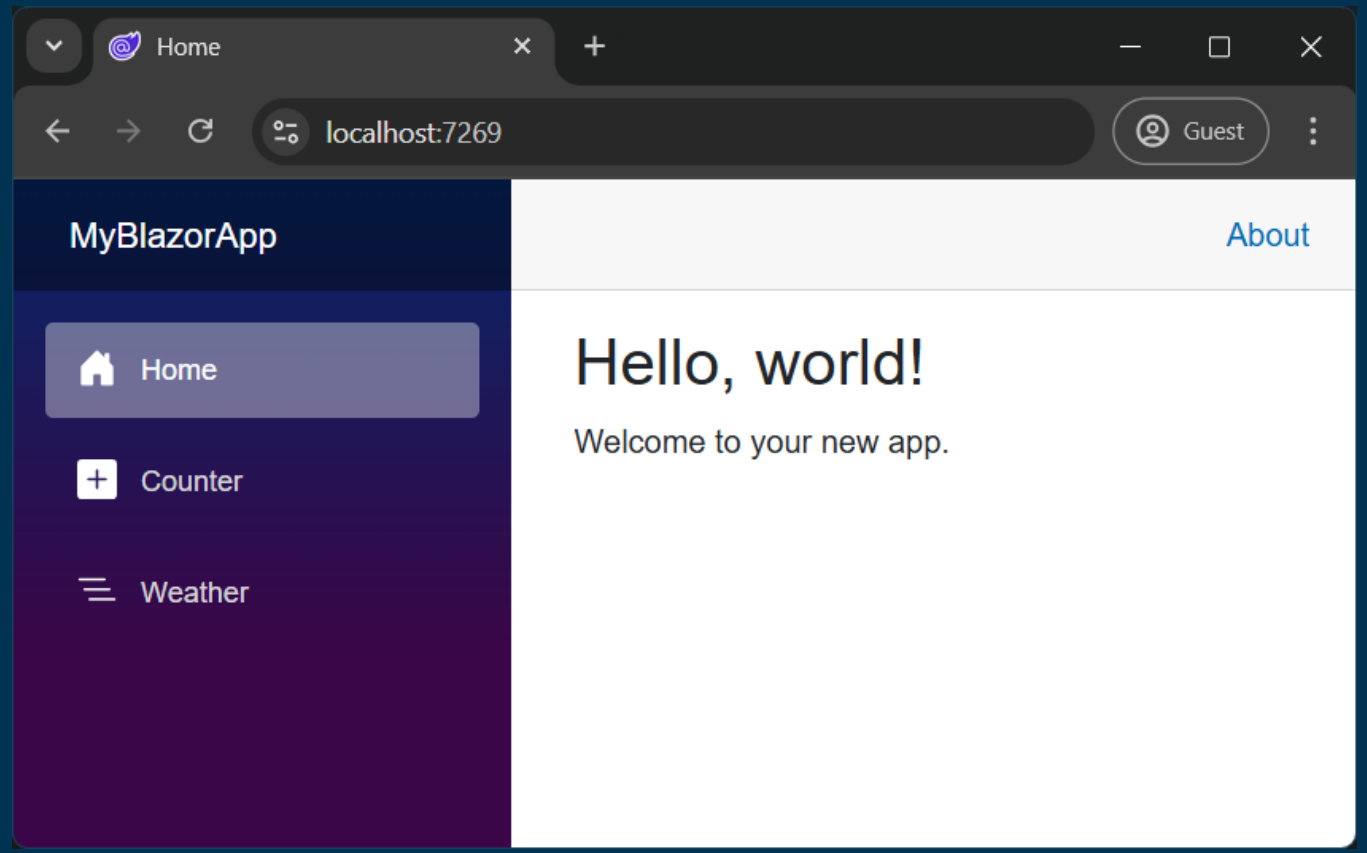
- If you get these SSL Certificate and Security Warning messages, click Yes.





Run the App

- The displayed page is defined by the Home.razor file located inside the Components/Pages directory.

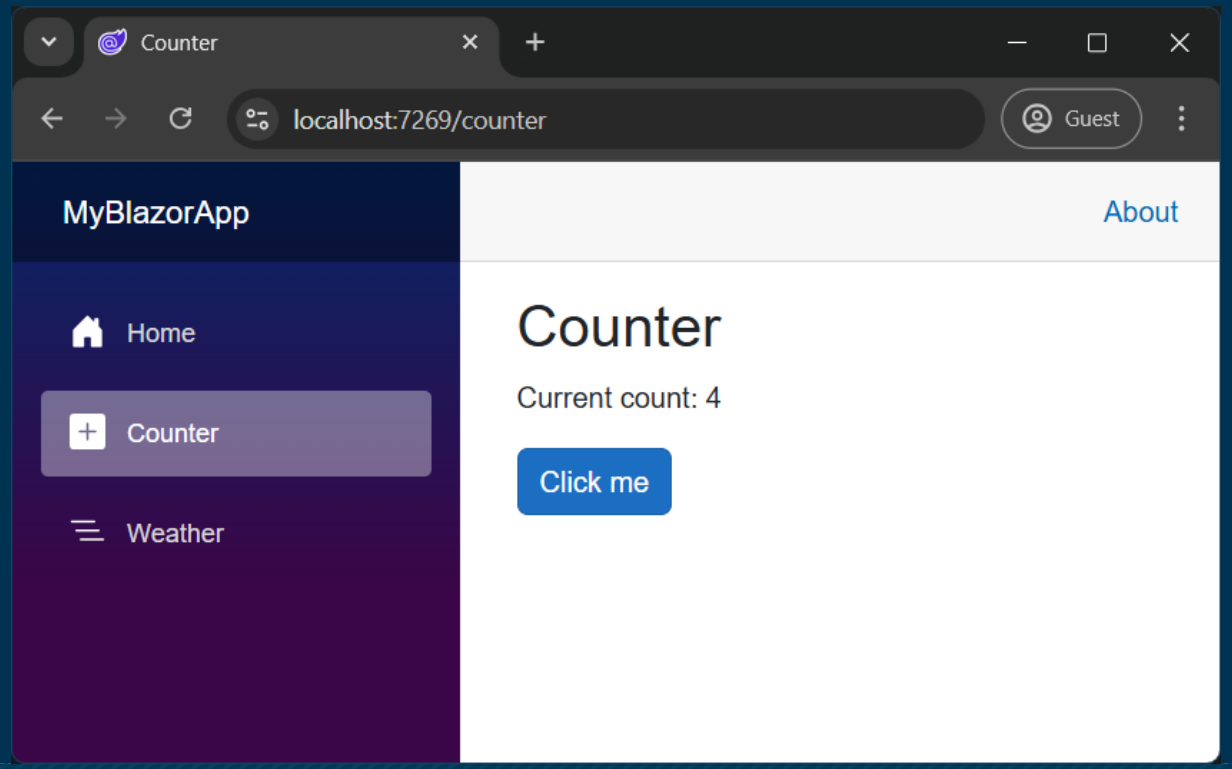



```razor
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.
```

Home.razor

Run the App

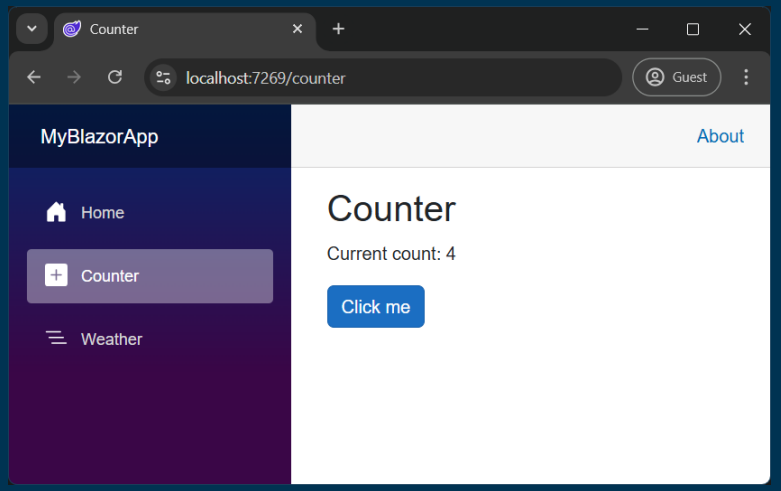
- In the running app, navigate to the Counter page by clicking the Counter tab in the sidebar on the left.
- Select the Click me button to increment the count without a page refresh.
- Incrementing a counter in a webpage normally requires writing JavaScript, but with Blazor you can use C#.



Run the App

- You can find the implementation of the Counter component at Counter.razor file located inside the Components/Pages directory.

```razor
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

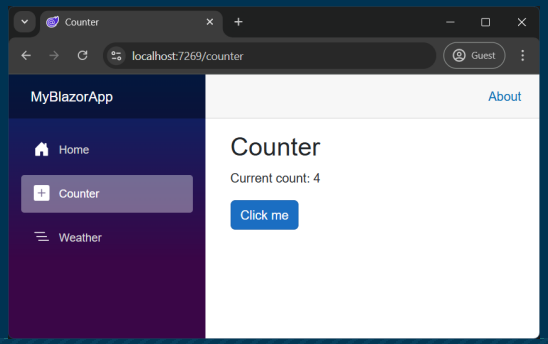


Run the App

- A request for /counter in the browser, as specified by the `@page` directive at the top, causes the Counter component to render its content.
- The `@rendermode` directive enables interactive server rendering for the component, so that it can handle user interface events from the browser.

```
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

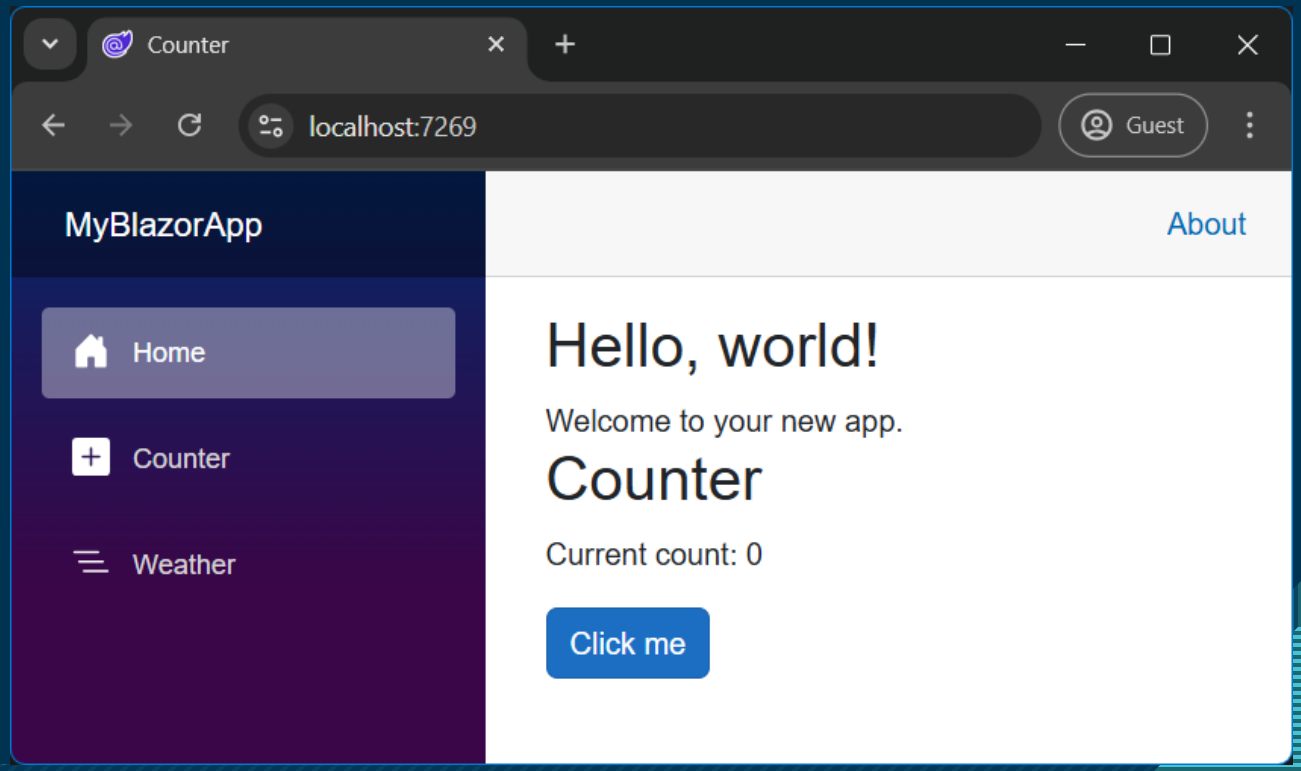
- Each time the Click me button is selected:
  - The `onclick` event is fired.
  - The `IncrementCount` method is called.
  - The `currentCount` is incremented.
  - The component is rendered to show the updated count.



Razor Component are Reusable

- Each of the .razor files defines a UI component that can be reused.
- Open the Home.razor file in Visual Studio, located in the Components/Pages folder.
- Add a Counter component to the app's homepage by adding a `<Counter />` element at the end of the Home.razor file.

```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<Counter />
```



Parameters in Razor Component

- Component parameters are specified using attributes, which allow you to set properties on the child component.
- Define a parameter on the Counter component for specifying how much it increments with every button click.
  - Add a public property for `IncrementAmount` with a `[Parameter]` attribute.
  - Change the `IncrementCount` method to use the `IncrementAmount` when incrementing the value of `currentCount`.

```
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    [Parameter]
    public int IncrementAmount { get; set; } = 1;

    private void IncrementCount()
    {
        currentCount += IncrementAmount;
    }
}
```

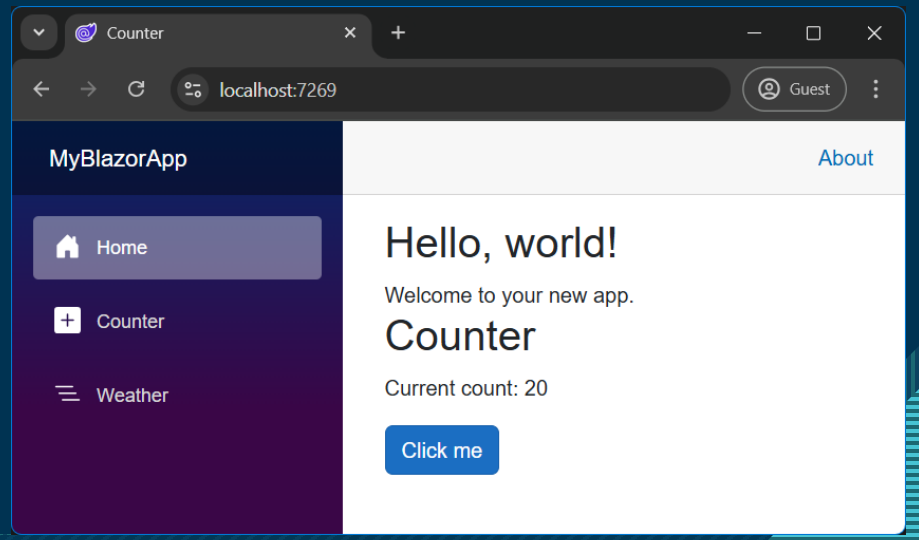
Parameters in Razor Component

- In Home.razor, update the `<Counter>` element to add an `IncrementAmount` attribute that changes the increment amount to ten.

- The Home component now has its own counter that increments by ten each time the Click me button is selected.

- The Counter component (Counter.razor) at /counter still increments by one.

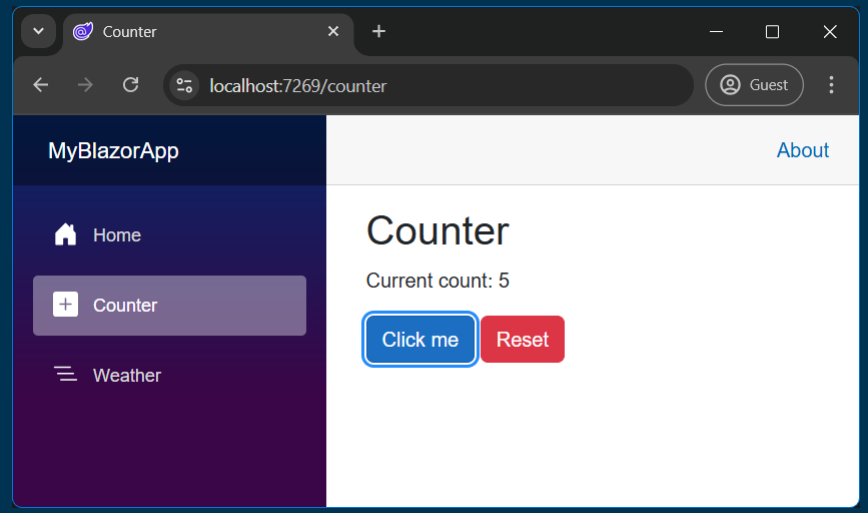
```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<Counter IncrementAmount="10" />
```

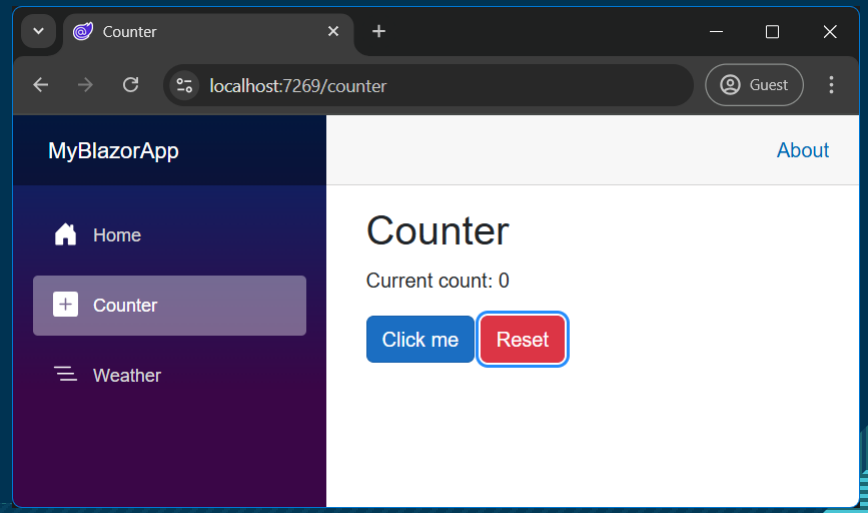


Exercise - Reset the Counter

- Modify the Counter component to allow users to reset the counter.

```razor
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

<button class="btn btn-danger" @onclick="ResetCount">Reset</button>

@code {
    private int currentCount = 0;

    [Parameter]
    public int IncrementAmount { get; set; } = 1;

    private void IncrementCount()
    {
        currentCount += IncrementAmount;
    }

    private void ResetCount()
    {
        currentCount = 0;
    }
}
```

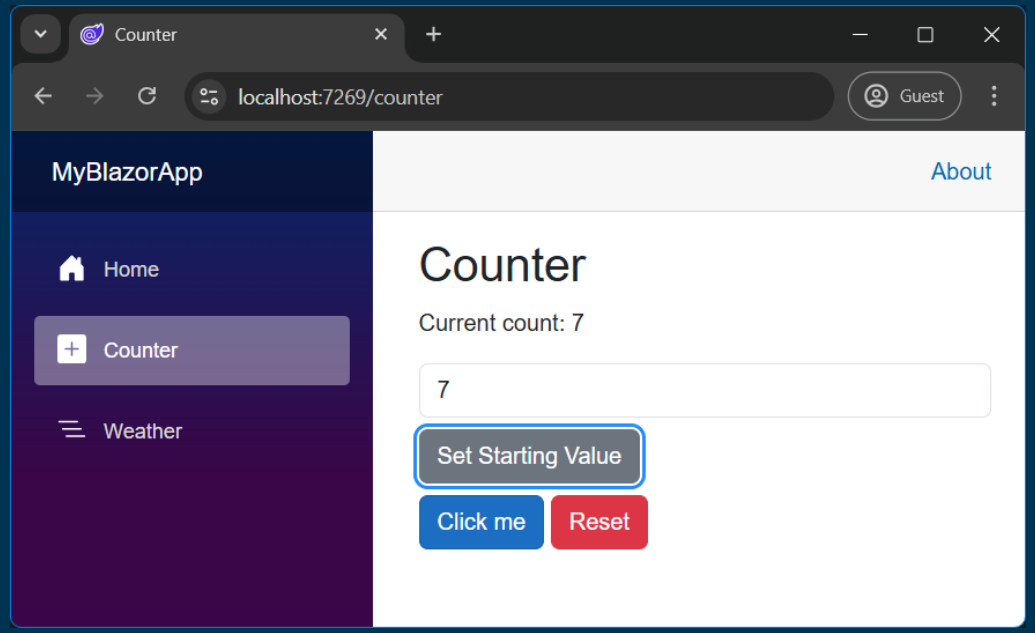




Exercise - Set the Starting Counter Value

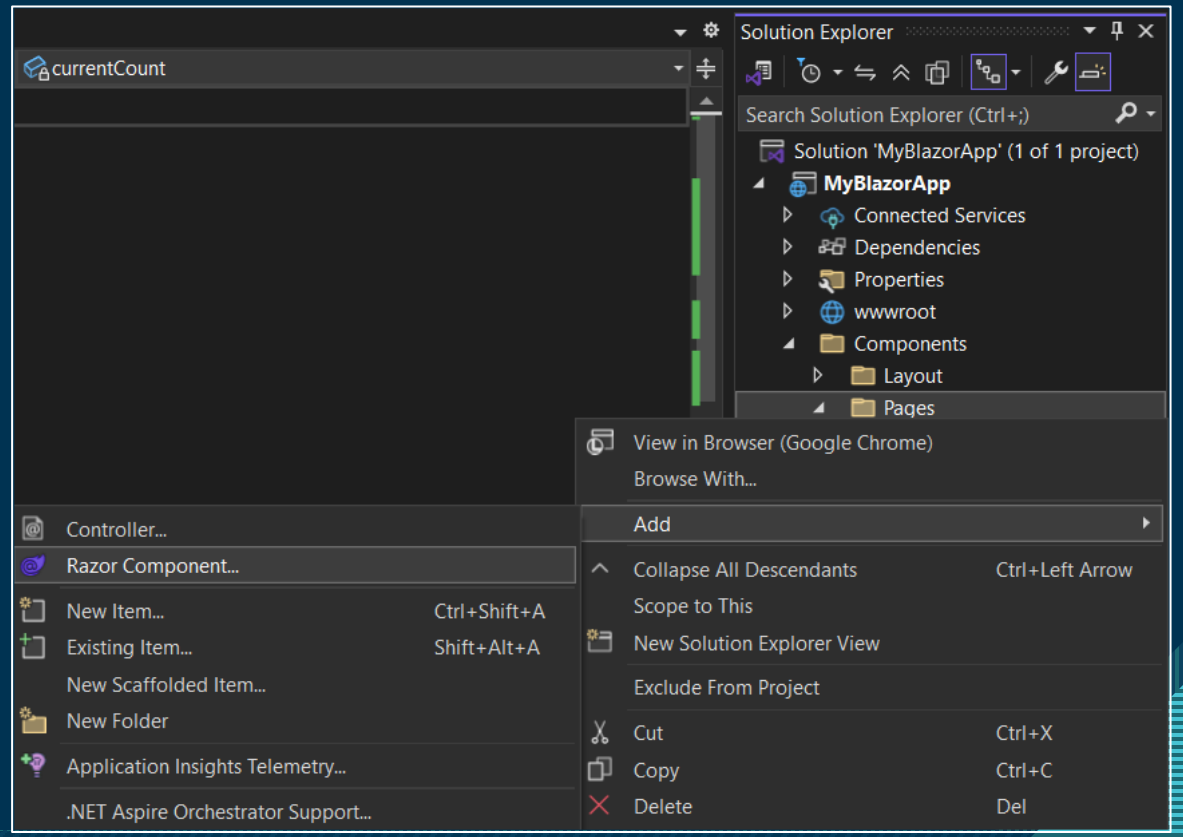
- Modify the Counter component to set a starting count using a text input.

```razor
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<div class="mb-2">
    <input type="number" @bind="startingValue" class="form-control" />
    <button class="btn btn-secondary mt-2" @onclick="SetStartingValue">Set Starting Value</button>
</div>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
<button class="btn btn-danger" @onclick="ResetCount">Reset</button>

@code {
    private int currentCount = 0;
    private int startingValue = 0;

    [Parameter]
    public int IncrementAmount { get; set; } = 1;

    private void IncrementCount()...

    private void ResetCount()
    {
        currentCount = 0;
        startingValue = 0;
    }

    private void SetStartingValue()
    {
        currentCount = startingValue;
    }
}
```



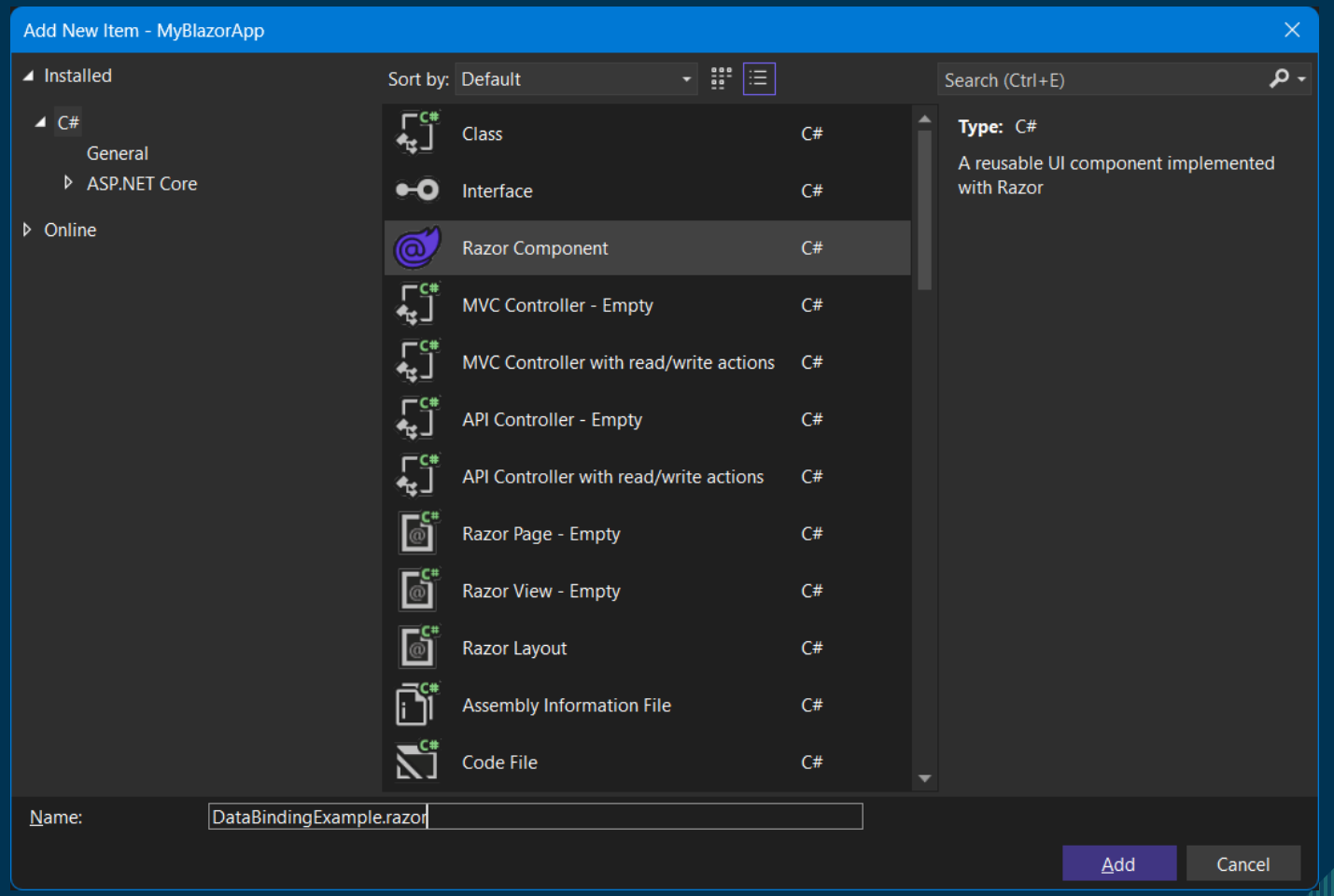
Data Binding in Blazor

- Data binding is when you link (or "bind") a variable in your C# code to something in the UI, like a text box.
- So, when the variable changes, the UI updates automatically, and vice versa.
- Let's say we want the user to type their name, and we display it right away.
- Add a new Razor Component by right-clicking on Pages folder.



Data Binding in Blazor

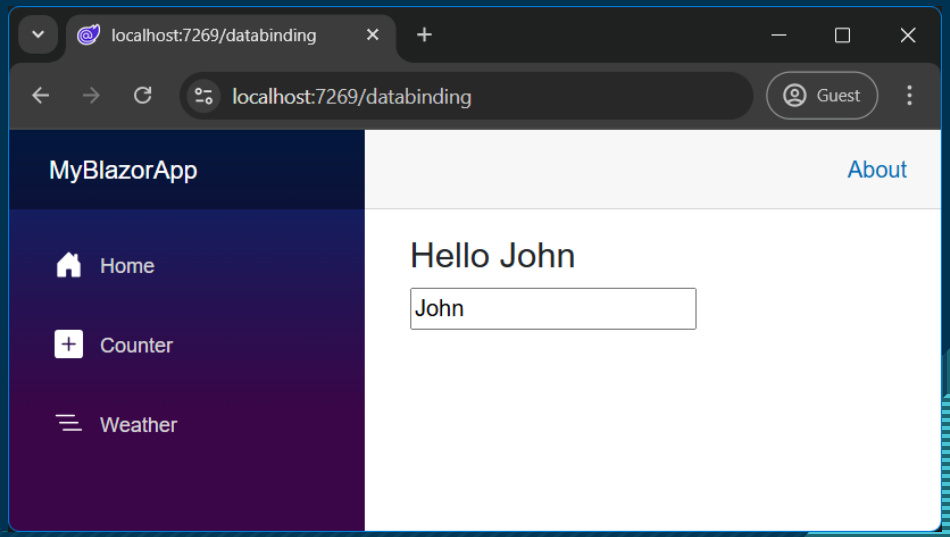
- Give it a meaningful name.
- *Razor component file names require a capitalized first letter.*
- The file name should be DataBindingExample.razor.



Data Binding in Blazor

- The input field is bound to the name variable.
- Whatever you type in the textbox updates name.
- Since @name is in the UI, it updates it instantly.
- *You need to click outside the textbox for the name to appear.*
- Two-Way Binding ⇔ Automatic Sync
  - UI → Variable
  - Variable → UI

```
@page "/databinding"
@rendermode InteractiveServer

<h3>Hello @name</h3>

<input type="text" @bind="name" placeholder="Type your name" />

@code {
    private string name = "";
}
```



Data Binding in Blazor

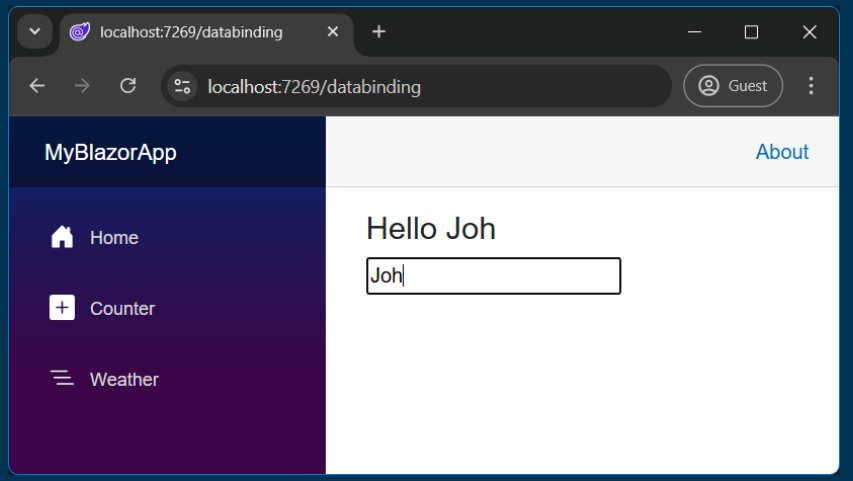
- By default, `@bind` updates the value when the input loses focus (`onchange` event).
- But you can make it update as you type by changing the event to `oninput`.
- Use:
  - `@bind-value="name"`
  - `@bind-value:event="oninput"`

```
@page "/databinding"
@rendermode InteractiveServer

<h3>Hello @name</h3>

<input type="text" @bind-value="name" @bind-value:event="oninput" placeholder="Type your name" />

@code {
    private string name = "";
}
```

Data Binding in Blazor

- `@bind-value="name"` tells Blazor to bind the input to the name variable.
- `@bind-value:event="oninput"` means:
  - Update the variable every time the user types, not just when they click away.

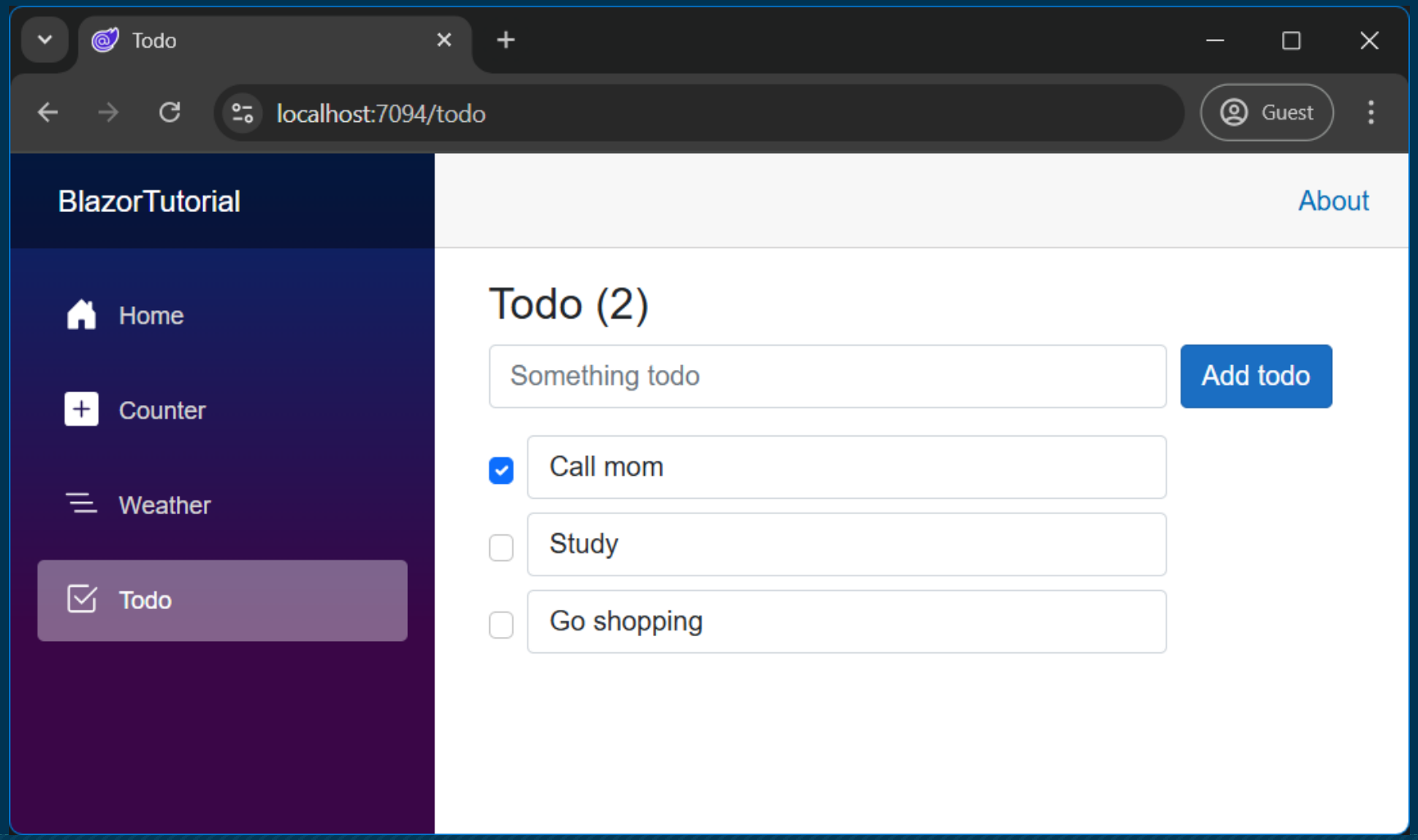
```razor
@page "/databinding"
@rendermode InteractiveServer

<h3>Hello @name</h3>

<input type="text" @bind-value="name" @bind-value:event="oninput" placeholder="Type your name" />

@code {
    private string name = "";
}
```



- `@bind` by itself is shorthand for `@bind-value` and defaults to `@bind-value:event="onchange"`.
- But if you want to change the event (like to `oninput`), you must use the full form: `@bind-value` and `@bind-value:event`.

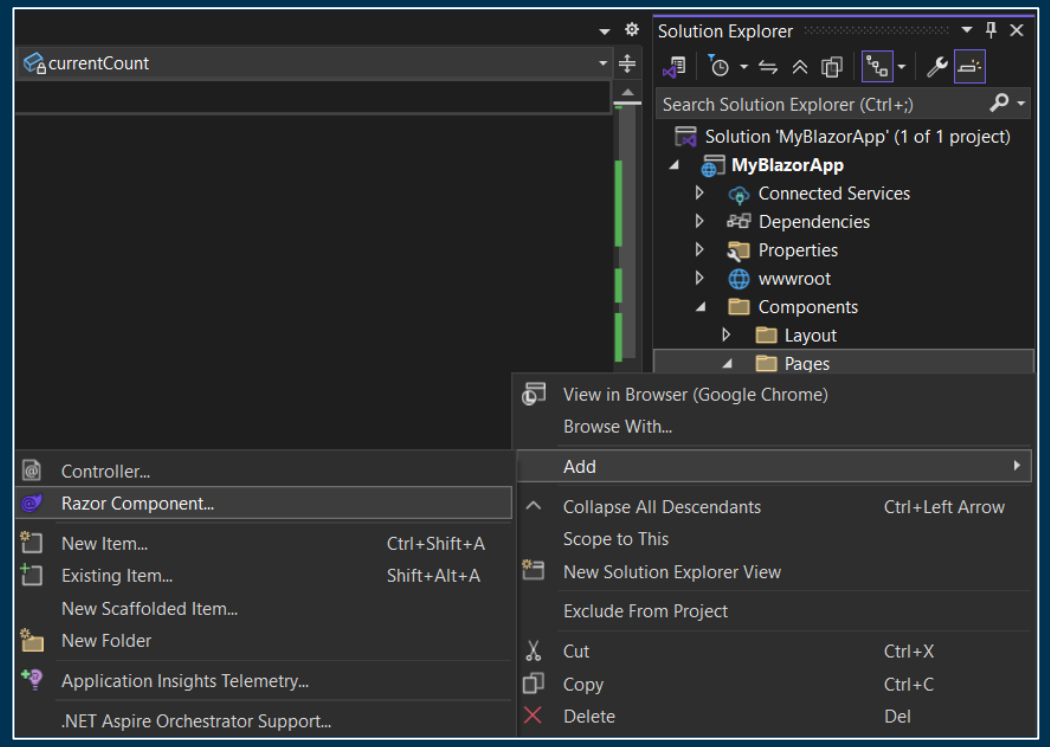
Exercise - Create a To-Do List

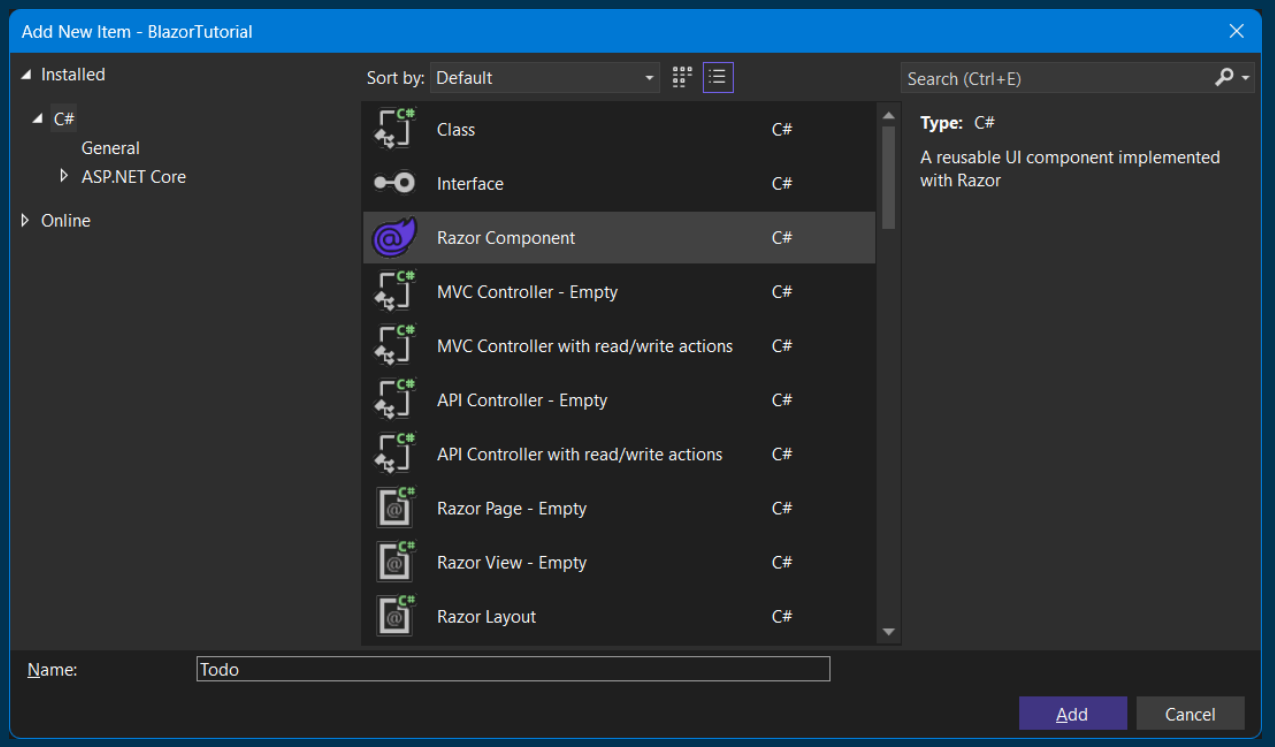
- Add a new Razor component to manage the Todo list.



Add a Razor Component

- Add a new Todo Razor component to the app.





- *Razor component file names require a capitalized first letter.*
- Make sure that Todo component file name starts with a capital letter T.

Add a Razor Component

- In the Todo component:
  - Add an `@page` Razor directive with a relative URL of /todo.
  - Enable interactivity on the page so that it isn't just statically rendered.
    - The Interactive Server render mode enables the component to handle UI events from the server.
  - Add a page title with the `PageTitle` component, which enables adding an HTML `<title>` element to the page.
- Save the Todo.razor file.

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

@code {

}
```

Add the Todo Component to the Navigation Bar

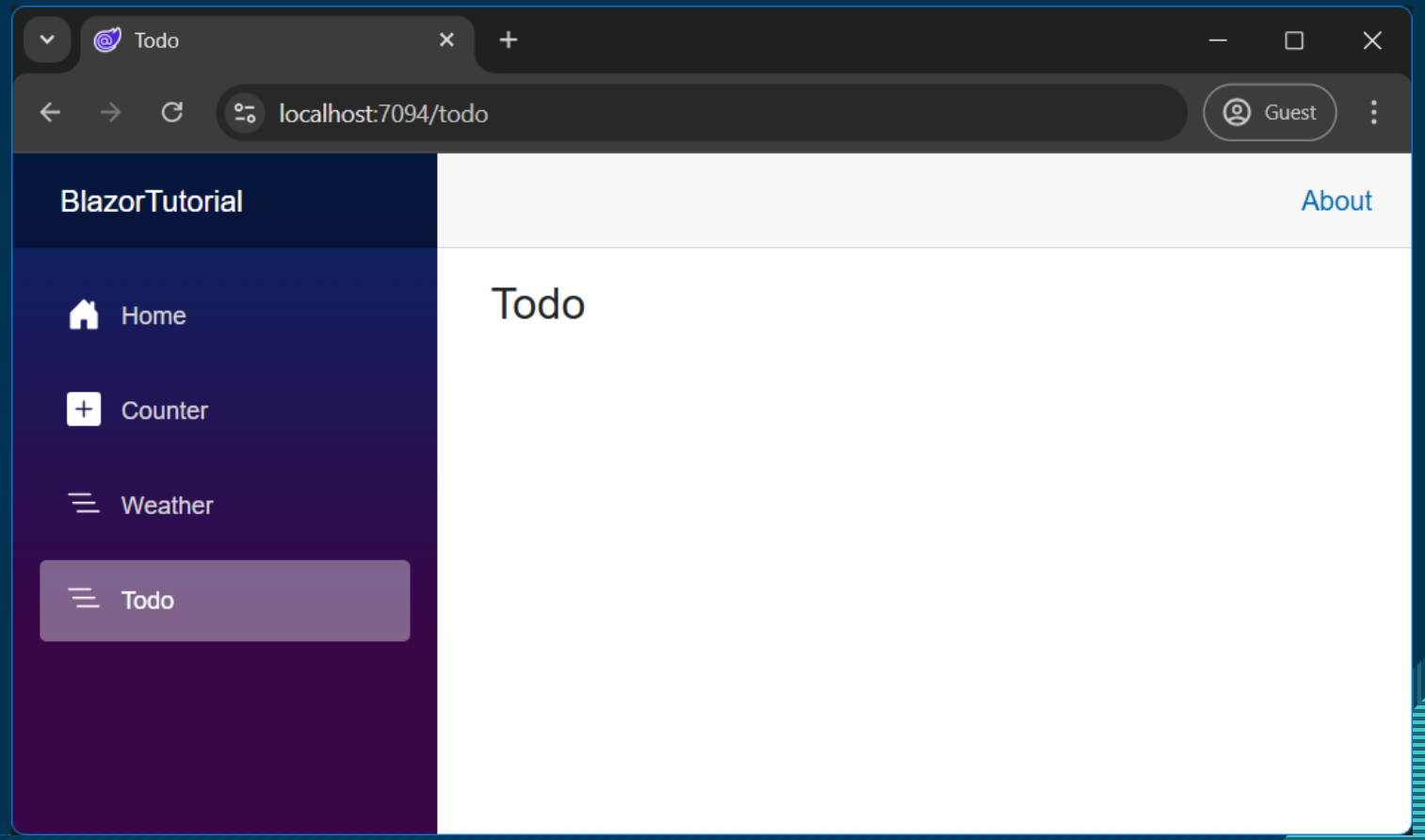
- The NavMenu component (inside the Components/Layout folder) is used in the app's layout.
- Layouts are components that allow you to avoid duplication of content in an app.
- The NavLink component provides a link in the app's UI when the URL is loaded by the app.
- In the navigation element (<nav>) of the NavMenu component, add the following element for the Todo component.

NavMenu.razor

```
        <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="todo">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Todo
            </NavLink>
        </div>
    </nav>
</div>
```

Run the App

- Run the app and visit the new Todo page by selecting the Todo link in the app's navigation bar, which loads the page at /todo.
- Leave the app running.
- Each time a file is saved, the app is automatically rebuilt, and the page in the browser is automatically reloaded.



Add a New Class

- Add a new class TodoItem.cs to the project to represents a todo item.

```
public class TodoItem
{
    0 references
    public string? Title { get; set; }
    0 references
    public bool IsDone { get; set; } = false;
}
```

Modify Todo.razor

- Return to the Todo component.
- Add a variable (field) for the todo items in the @code block.
- The Todo component uses this variable to maintain the state of the todo list.

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

@code {
    private List<TodoItem> todos = new List<TodoItem>();
}
```

Modify Todo.razor

- Add unordered list markup and a `foreach` loop to render each todo item as a list item (`<li>`).

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

@code {
    private List<TodoItem> todos = new List<TodoItem>();
}
```

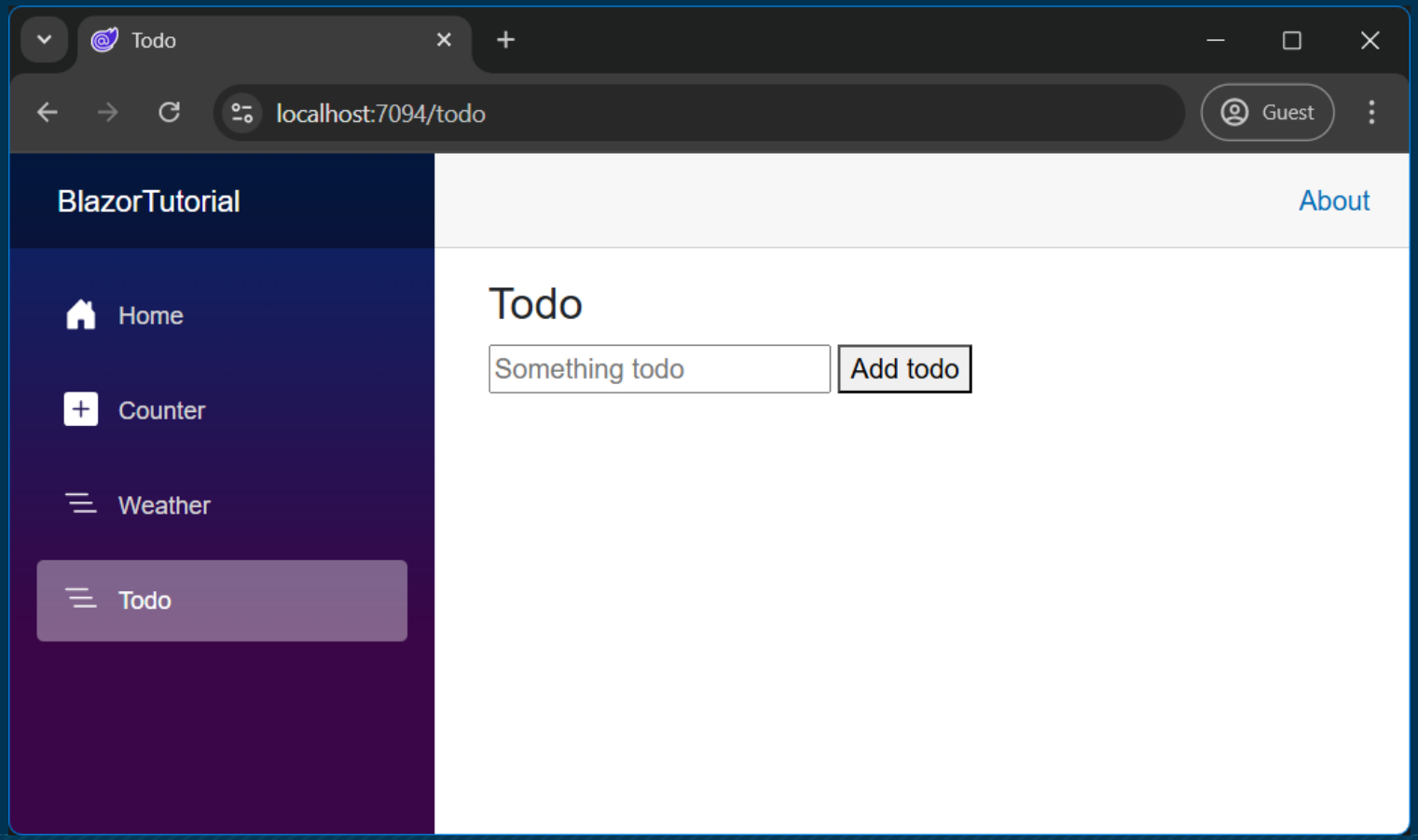
Modify Todo.razor

- The app requires UI elements for adding todo items to the list.
- Add a text input (`<input>`) and a button (`<button>`) above the unordered list (`<ul>...</ul>`).

```
<h3>Todo</h3>

<input type="text" placeholder="Something todo" />
<button>Add todo</button>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>
```

Run the App

- Save the changes.
- The app is automatically rebuilt and the browser reloads the page.



Modify Todo.razor

- When the Add todo button is clicked, nothing happens because an event handler isn't attached to the button.
- Add an `AddTodo` method to the Todo component.
- Also, to get the title of the new todo item, add a `newTodo` string field to the `@code` block.

```
@code {
    private List<TodoItem> todos = new List<TodoItem>();
    private string? newTodo;

    private void AddTodo()
    {
        // Todo: Add the todo
    }
}
```

Modify Todo.razor

- Modify the text `<input>` element to bind `newTodo` with the `@bind` attribute.
- And register the `AddTodo` method for the button using the `@onclick` attribute.

```
<h3>Todo</h3>

<input type="text" placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>
```

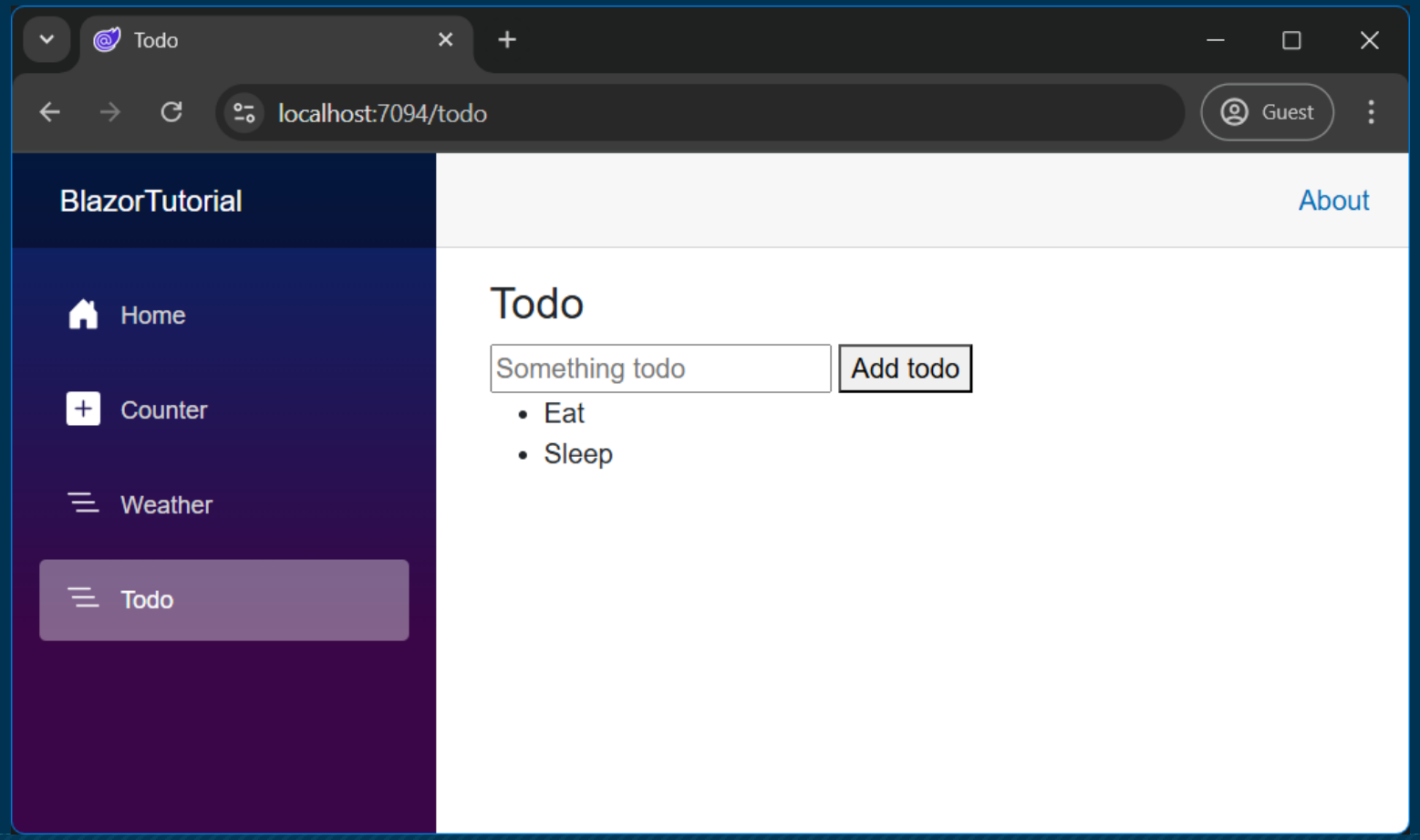
Modify Todo.razor

- Update the `AddTodo` method to add the `TodoItem` with the specified title to the list.
- Clear the value of the text input by setting `newTodo` to an empty string.

```
@code {
    private List<TodoItem> todos = new List<TodoItem>();
    private string? newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            TodoItem todoItem = new TodoItem { Title = newTodo };
            todos.Add(todoItem);
            newTodo = string.Empty;
        }
    }
}
```

Run the App

- Save the Todo.razor file.
- The app is automatically rebuilt and the page reloads in the browser.



Modify Todo.razor

- The title text for each todo item can be made editable, and a checkbox can help the user keep track of completed items.
- Add a checkbox input for each todo item and bind its value to the `IsDone` property.
- Change `@todo.Title` to an `<input>` element bound to `todo.Title` with `@bind`.

```
<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input type="text" @bind="todo.Title" />
        </li>
    }
</ul>
```

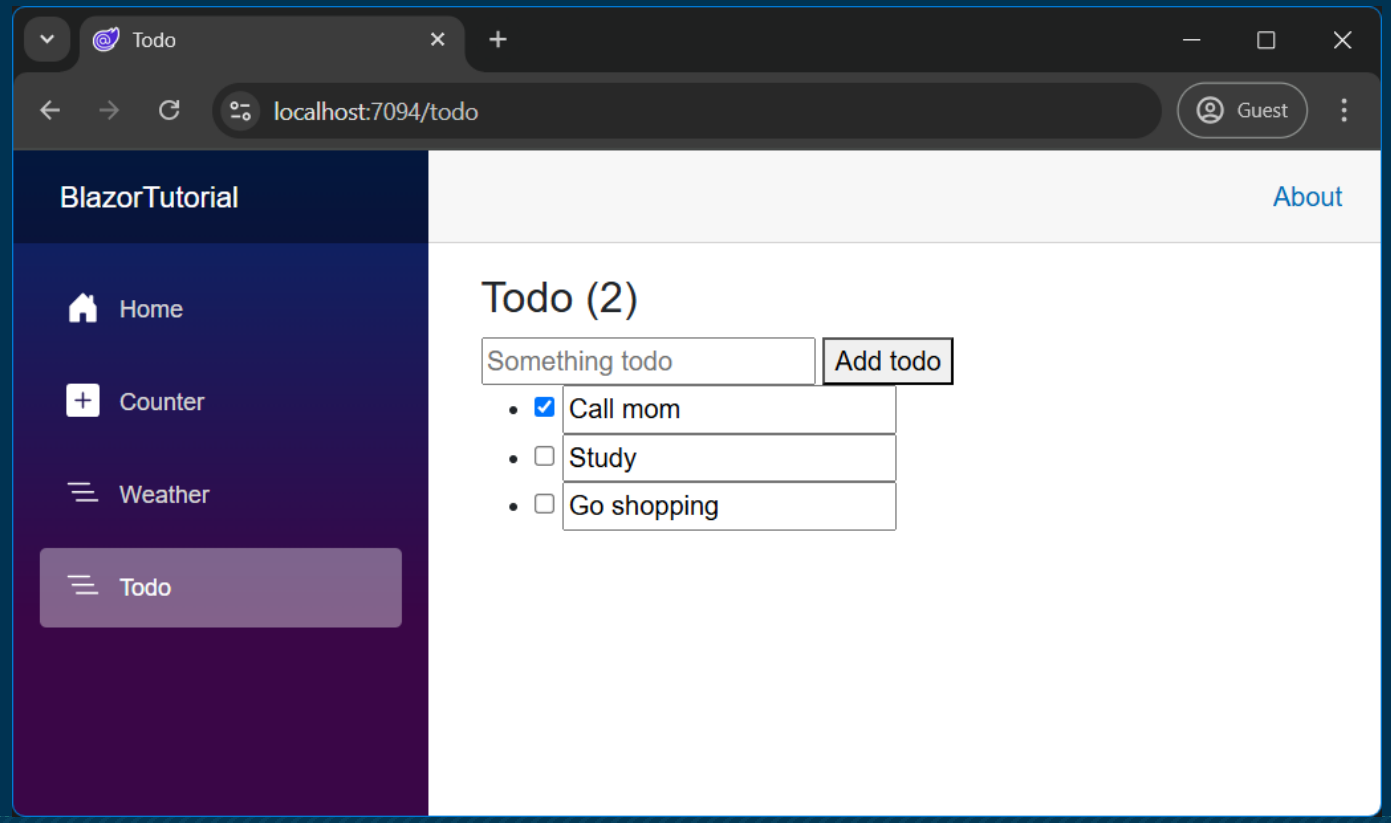
Modify Todo.razor

- Update the `<h3>` header to show a count of the number of todo items that aren't complete (`IsDone` is `false`).
- The Razor expression in the following header evaluates each time Blazor re-renders the component.

```
<PageTitle>Todo</PageTitle>

<h3>Todo (@todos.Count(todo => !todo.IsDone)) </h3>

<input type="text" placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>
```

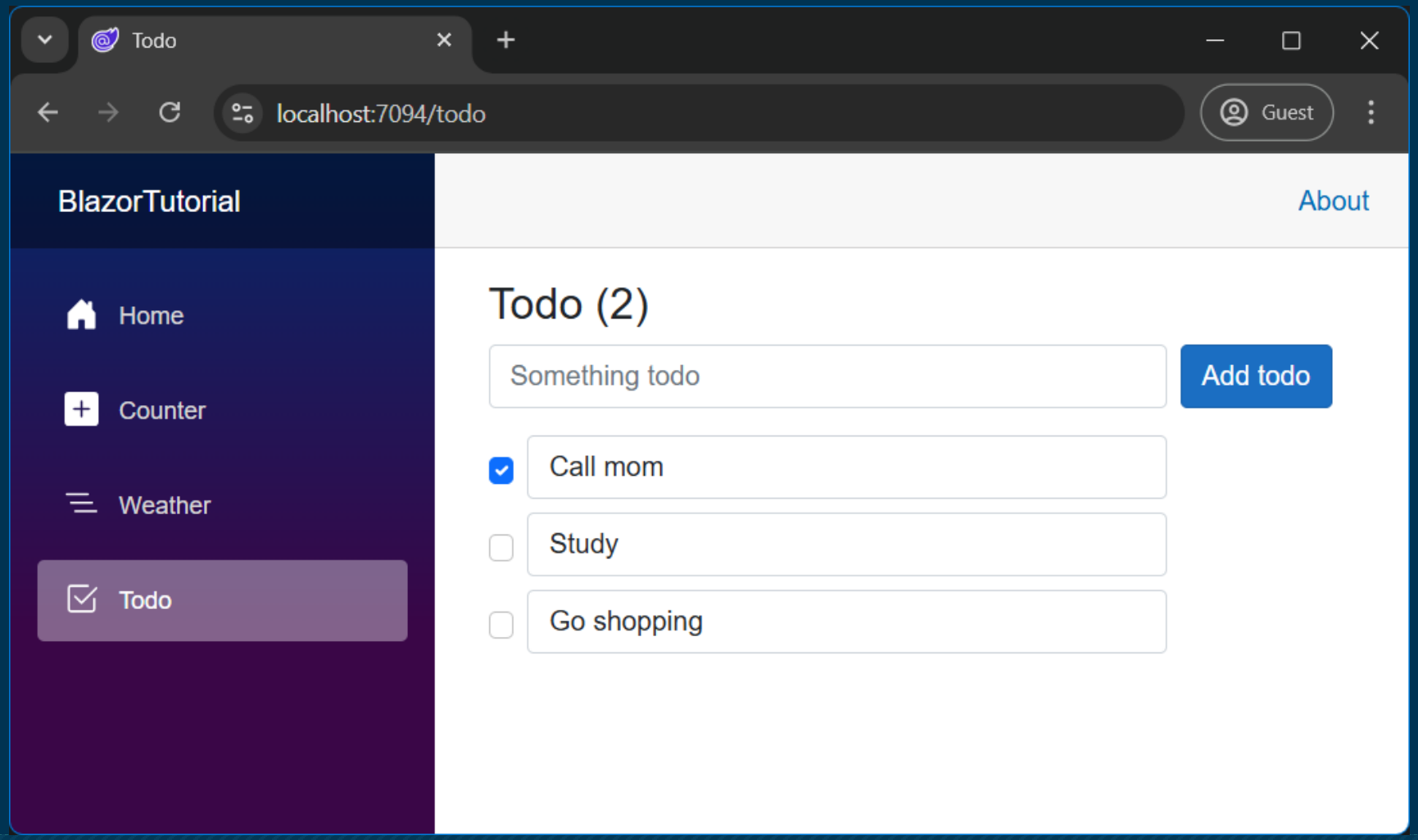
Run the App

- Save the Todo.razor file.
- The app is automatically rebuilt and the page reloads in the browser.
- Add items, edit items, and mark todo items done to test the component.



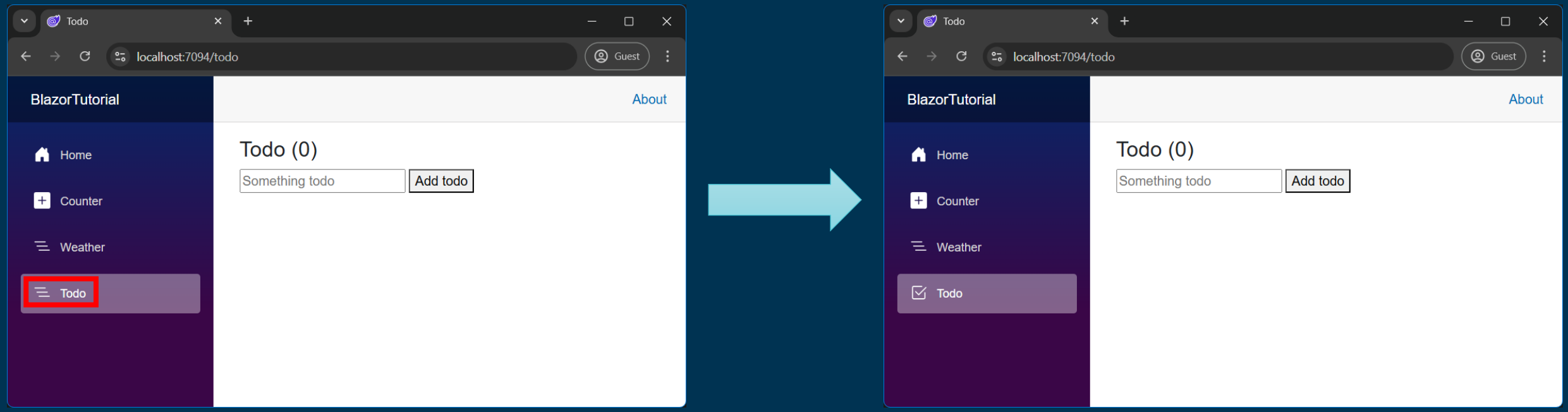
Make the App Look Modern

- Now let's make that app look clean and modern by using Bootstrap.



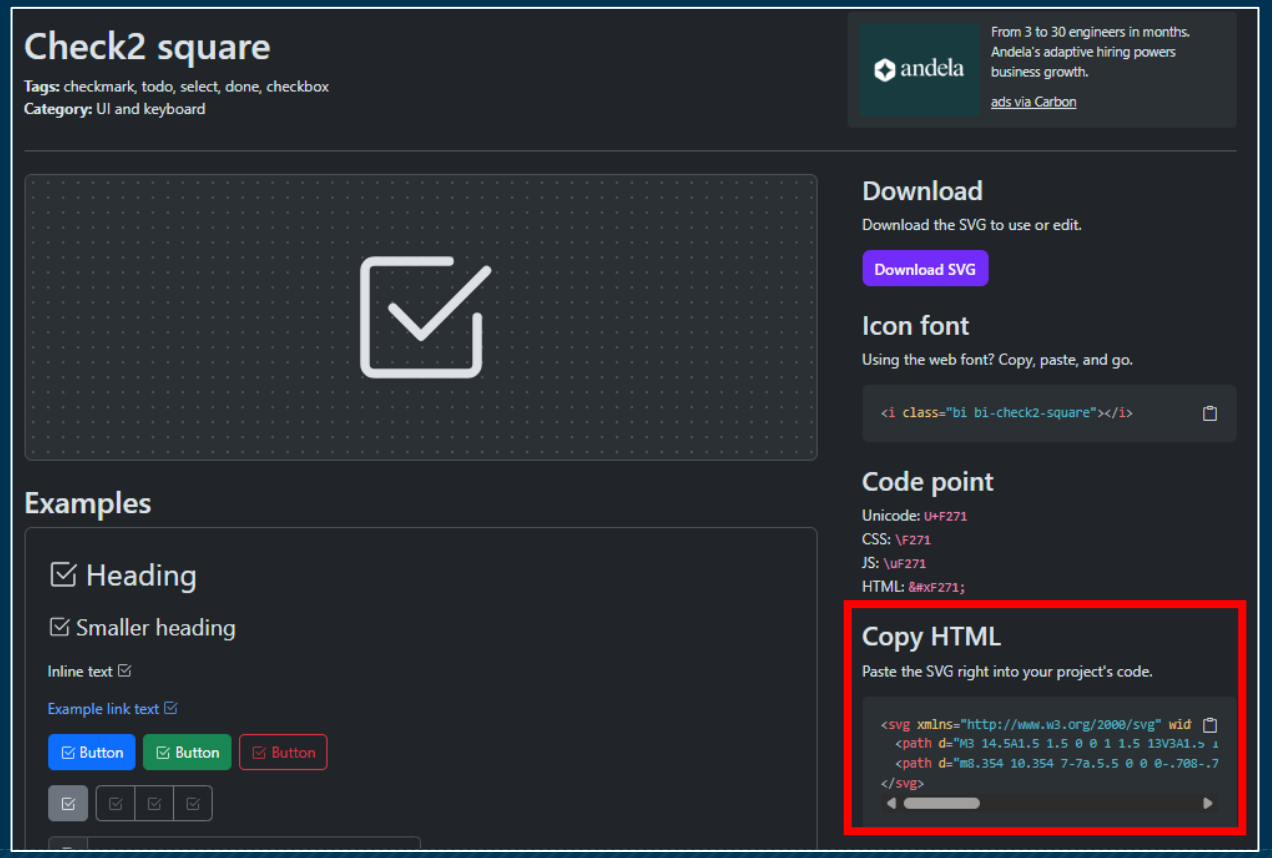
Change the Todo Icon

- First of all, let's change the Todo navigation's icon.



Change the Todo Icon

- Go to this link to get the icon's HTML code.
  - https://icons.getbootstrap.com/icons/check2-square/
- Copy the `<svg>` HTML code.



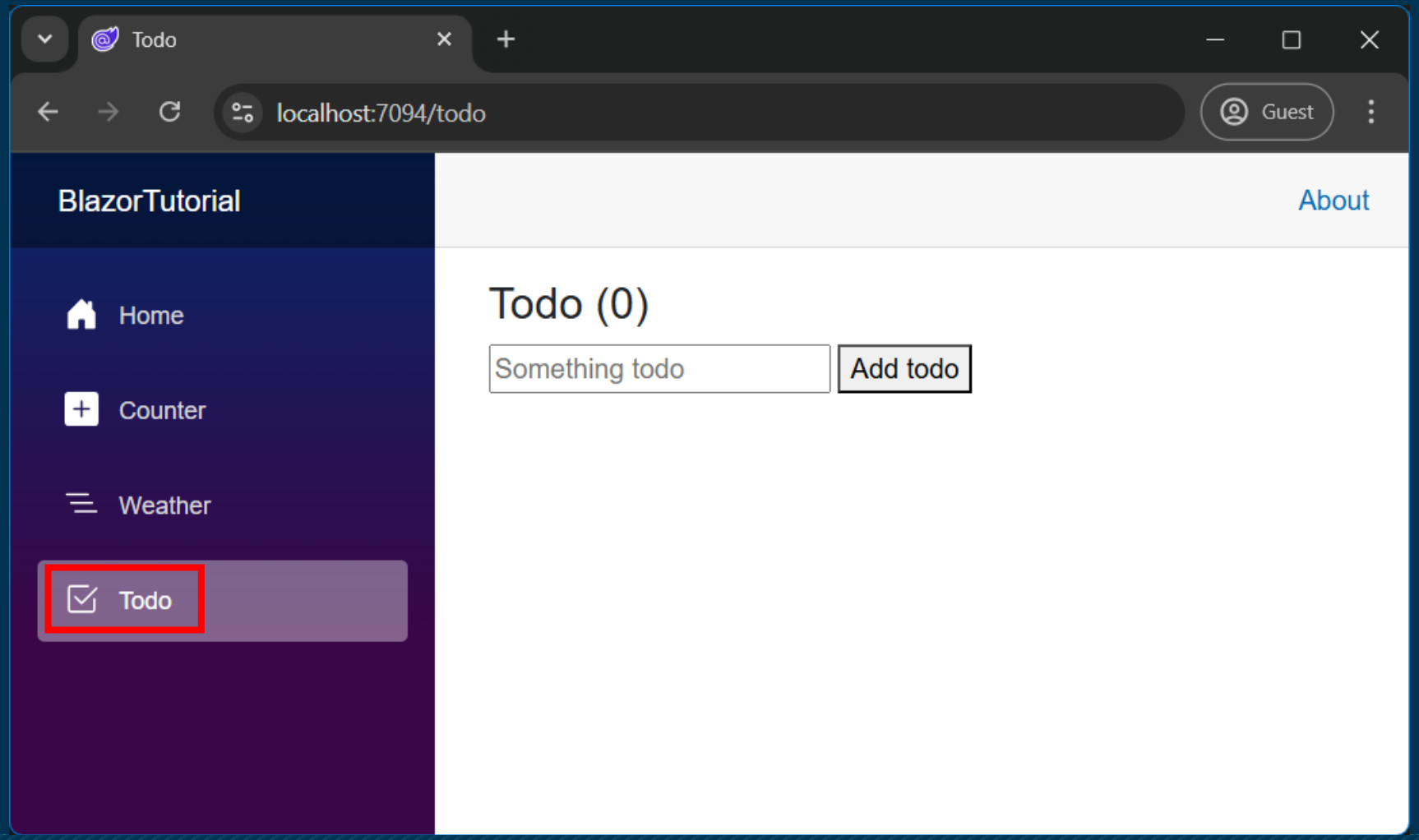
Change the Todo Icon

- Open NavMenu.razor.
- Remove the element and paste the copied <svg> code.

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="todo">
        <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16"
          fill="currentColor" class="bi bi-check2-square" viewBox="0 0 16 16">
            <path d="M3 14.5A1.5 1.5 0 0 1 1.5 13V3A1.5 1.5 0 0 1 3 1.5h8a.5.5 0 0 1 0
              1H3a.5.5 0 0 0-.5.5v10a.5.5 0 0 0 .5.5h10a.5.5 0 0 0 .5-.5V8a.5.5 0 0 1 1
              0v5a1.5 1.5 0 0 1-1.5 1.5z" />
            <path d="m8.354 10.354 7-7a.5.5 0 0 0-.708-.708L8 9.293 5.354 6.646a.5.5 0
              1 0-.708.708l3 3a.5.5 0 0 0 .708 0" />
        </svg> Todo
    </NavLink>
</div>
```

Run the App

- Save the NavMenu.razor file and check the output.



Apply Bootstrap

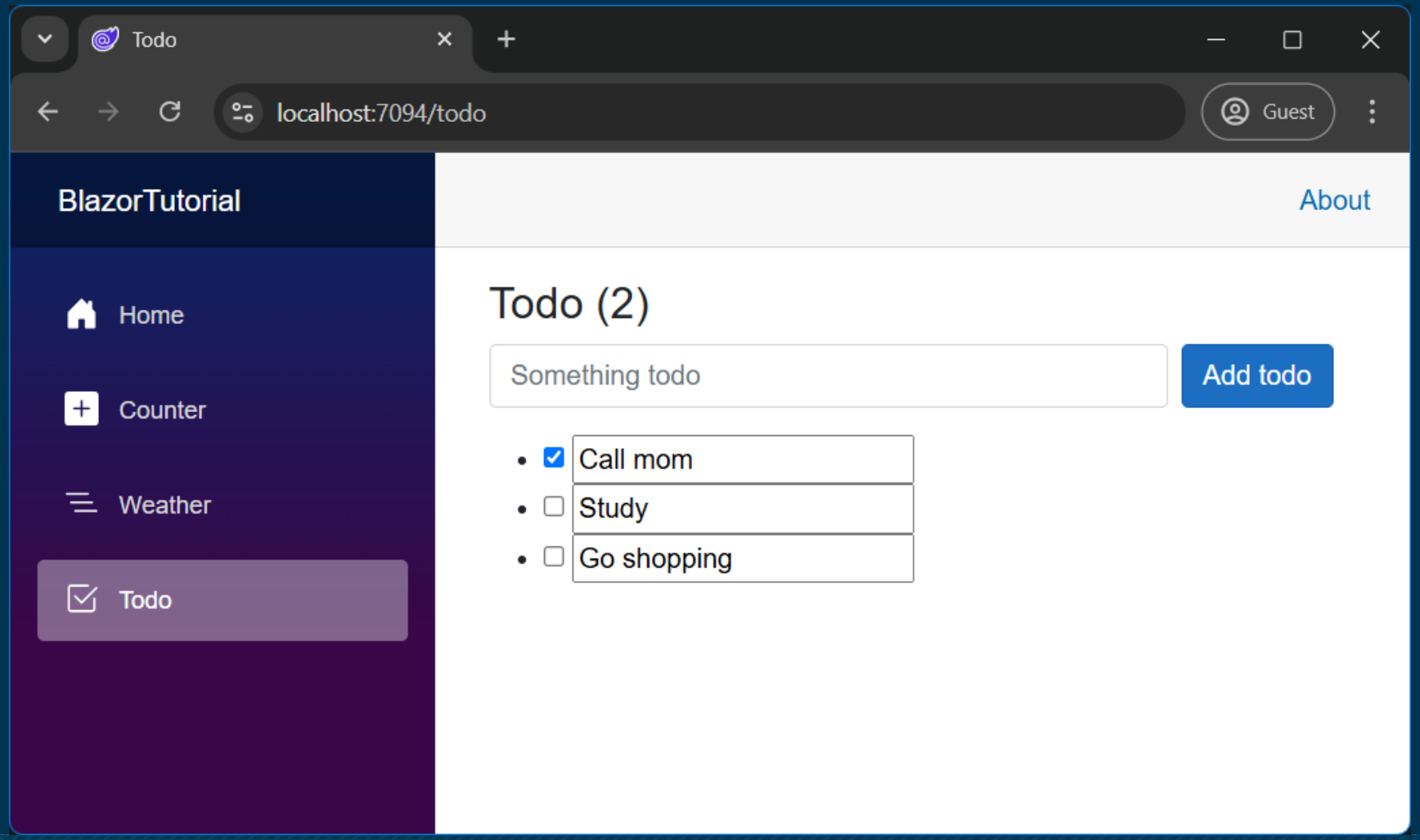
- Go back to Todo.razor.
- Enclose the input and button elements in a `<div>`.
- Apply Bootstrap classes to the `<div>`, `<input>` and `<button>`.

```
<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>

<div class="d-flex align-items-center mb-3">
    <input type="text" placeholder="Something todo"
           class="form-control me-2" style="max-width: 400px;"
           @bind="newTodo" />
    <button class="btn btn-primary" @onclick="AddTodo">Add todo</button>
</div>
```

- `d-flex` → (`display: flex`) puts input and button side by side.
- `align-items-center` → vertically aligns the button and input.
- `mb-3` → adds bottom margin to separate the div from the list.
- `me-2` → adds some right margin to the input for spacing.

Run the App

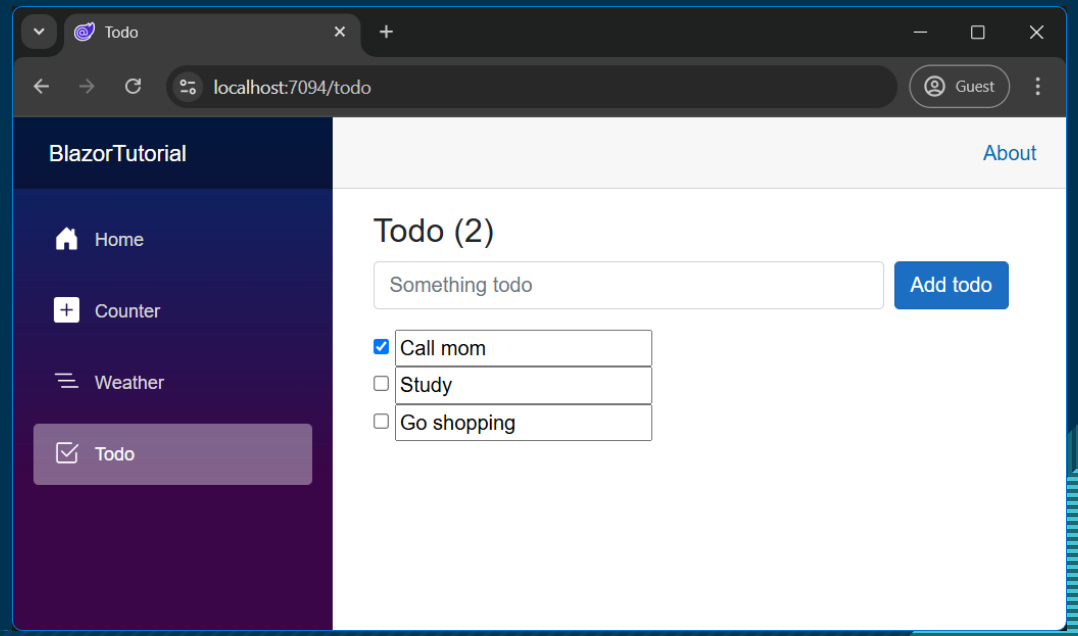
- Save the Todo.razor file and check the output.



Apply Bootstrap

- Now, let's update the `<ul>` and make it look modern as well.
- Apply class `list-unstyled` to the `<ul>` to get rid off the bullet points.

```
<ul class="list-unstyled">
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input type="text" @bind="todo.Title" />
        </li>
    }
</ul>
```



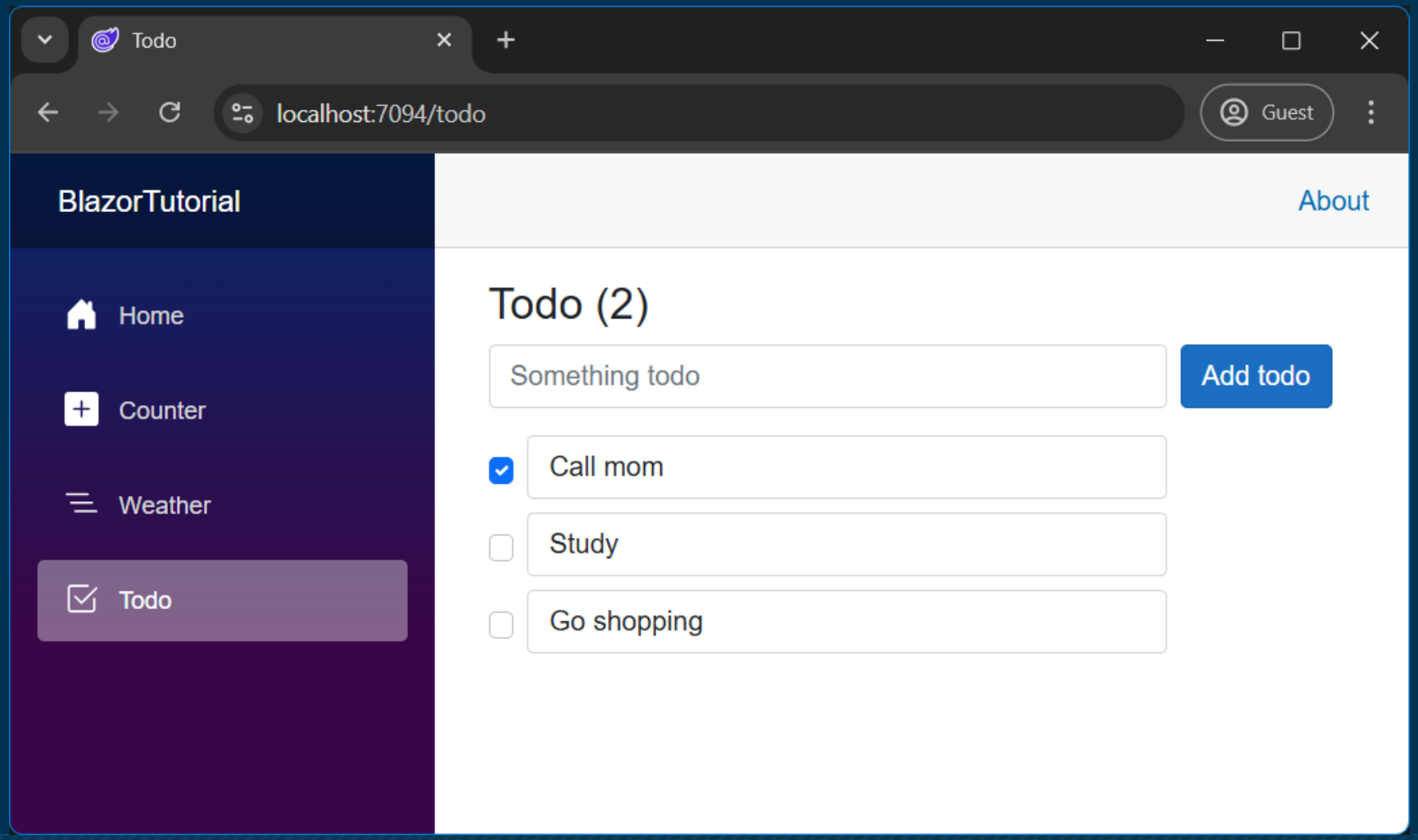
Apply Bootstrap

- Apply Bootstrap classes to the `<li>` and the `<input>` elements contained within it.

```
<ul class="list-unstyled">
    @foreach (var todo in todos)
    {
        <li class="d-flex align-items-center mb-2" style="max-width: 400px;">
            <input type="checkbox" class="form-check-input me-2" @bind="todo.IsDone" />
            <input type="text" class="form-control" @bind="todo.Title" />
        </li>
    }
</ul>
```

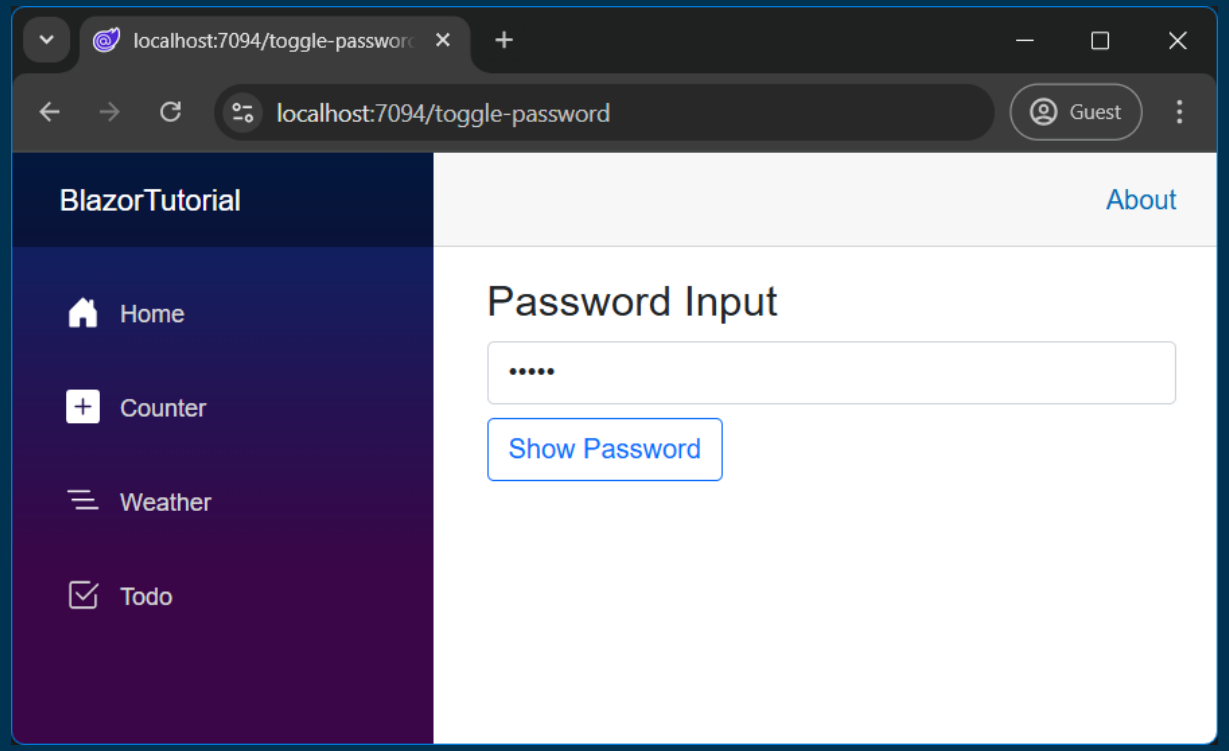
Run the App

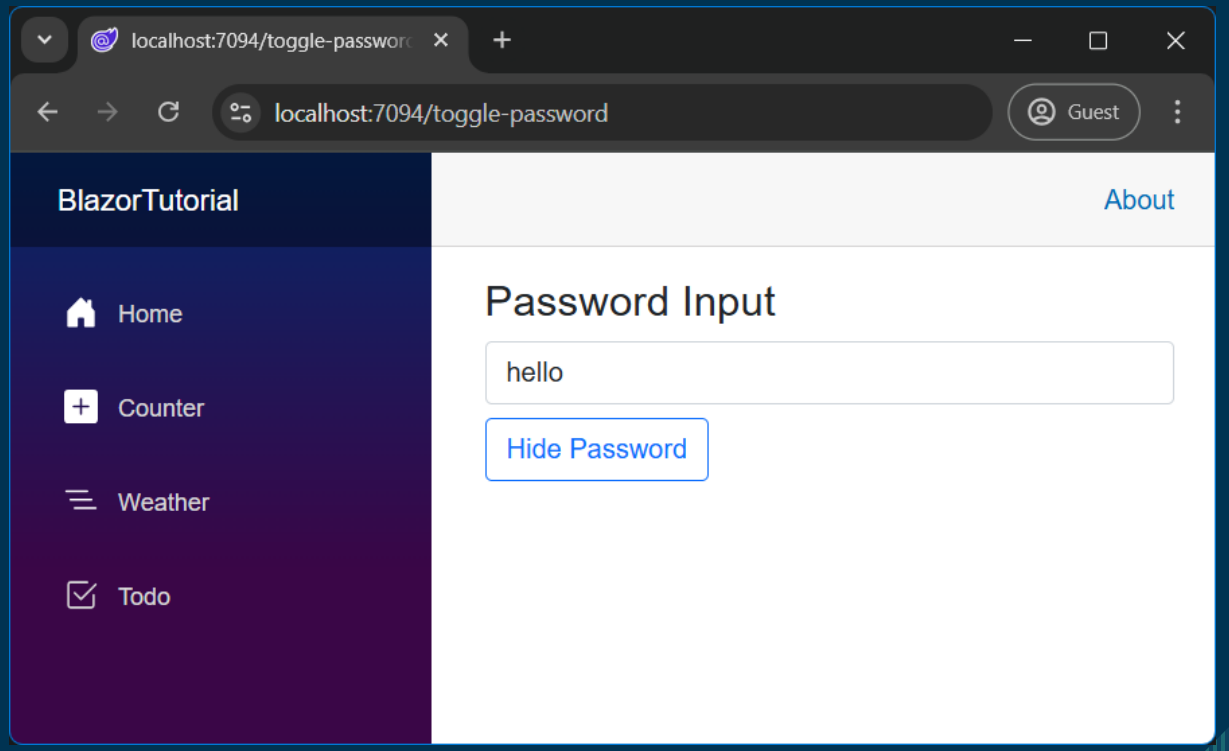
- Save the Todo.razor file and check the output.



Do It Yourself!

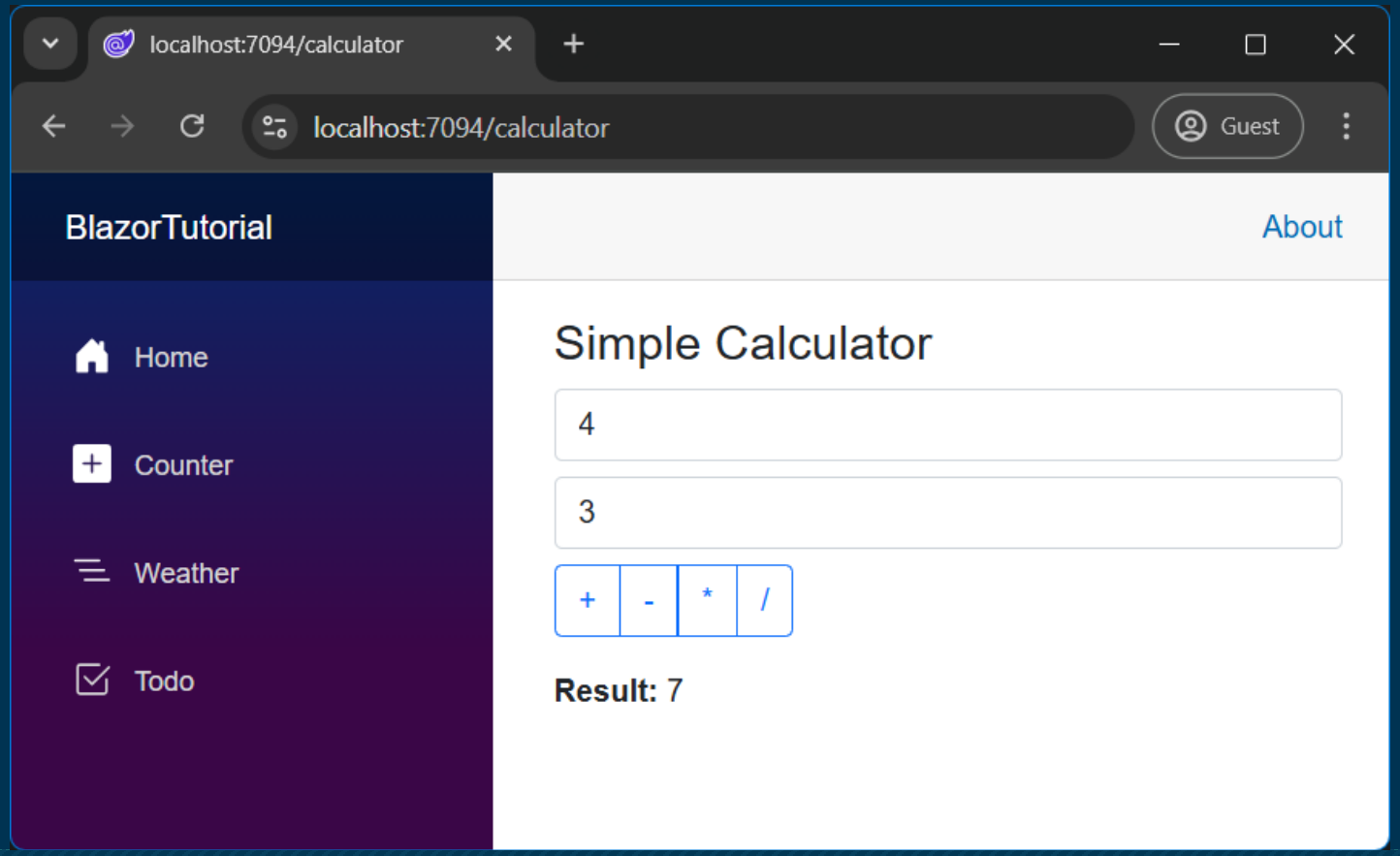
- **Show/Hide Password Toggle**:
  - Create a password box with a toggle button to show or hide the password.





Do It Yourself!

- **Simple Calculator**:
  - Build a basic calculator that lets users input two numbers and perform add, subtract, multiply, or divide operations.



Thank You

References

Some material has been taken from:

- ASP.NET Core Blazor:
  - https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor
- Introduction to Web Development with Blazor:
  - https://learn.microsoft.com/en-us/training/modules/blazor-introduction/2-what-is-blazor
- Build your first web app with ASP.NET Core using Blazor:
  - https://dotnet.microsoft.com/en-us/learn/aspnet/blazor-tutorial/intro
- Build a Blazor todo list app:
  - https://learn.microsoft.com/en-us/aspnet/core/blazor/tutorials/build-a-blazor-app
- [Book] – Murach's ASP.NET Core MVC, Chapter 1

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.