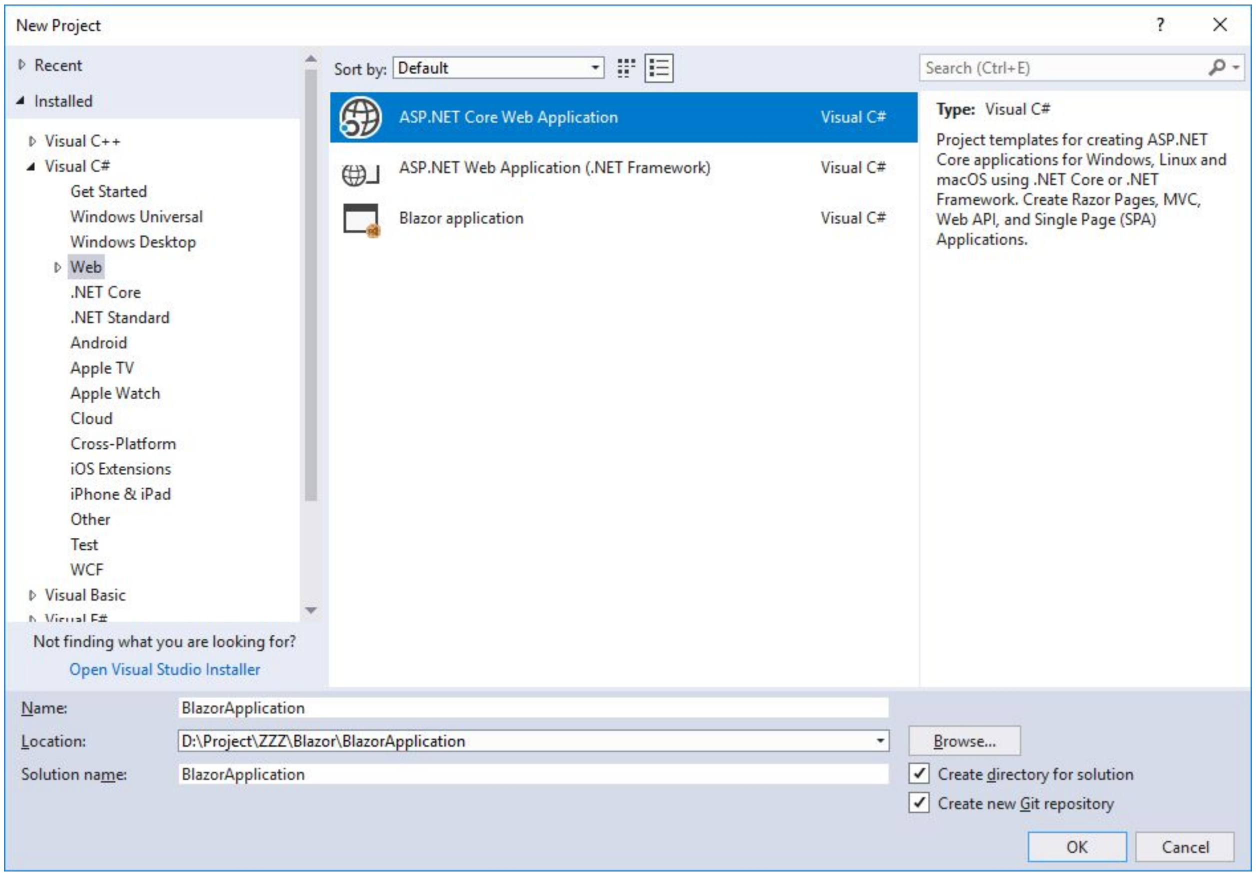
Application Development using .NET Core and Blazor Labs

Lab 1 Create Blazor Application

Blazor is an experimental .NET web framework using C#/Razor and HTML that runs in the browser with WebAssembly. Blazor provides all of the benefits of a client-side web UI framework using .NET on the client and optionally on the server.

Once the environment is set up, you can create the Blazor application from File > New menu and choose Project... menu item.

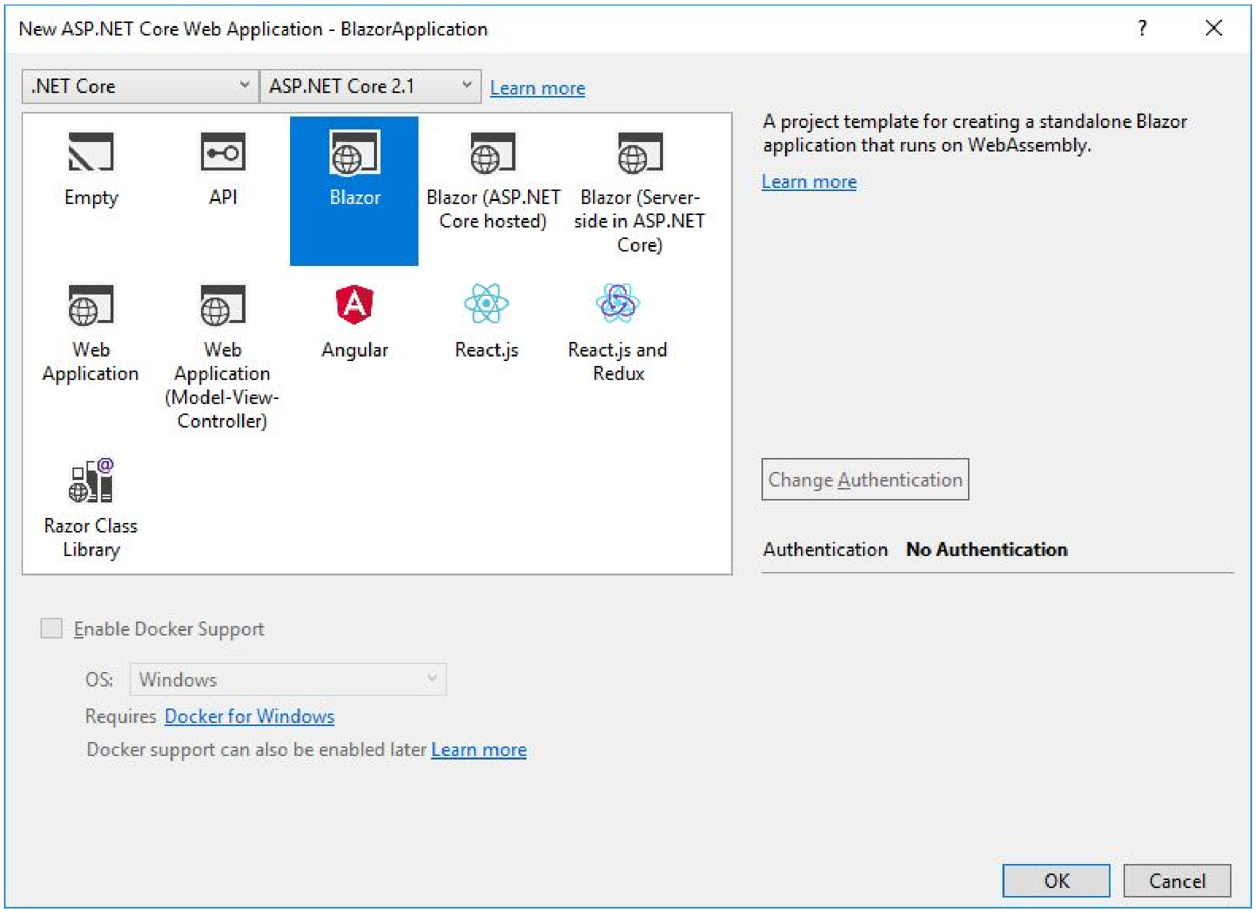


Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

Name: BlazorApplication
Location: D:\Project\ZZZ\Blazor\BlazorApplication
Solution name: BlazorApplication
[✓] Create directory for solution
[✓] Create new Git repository

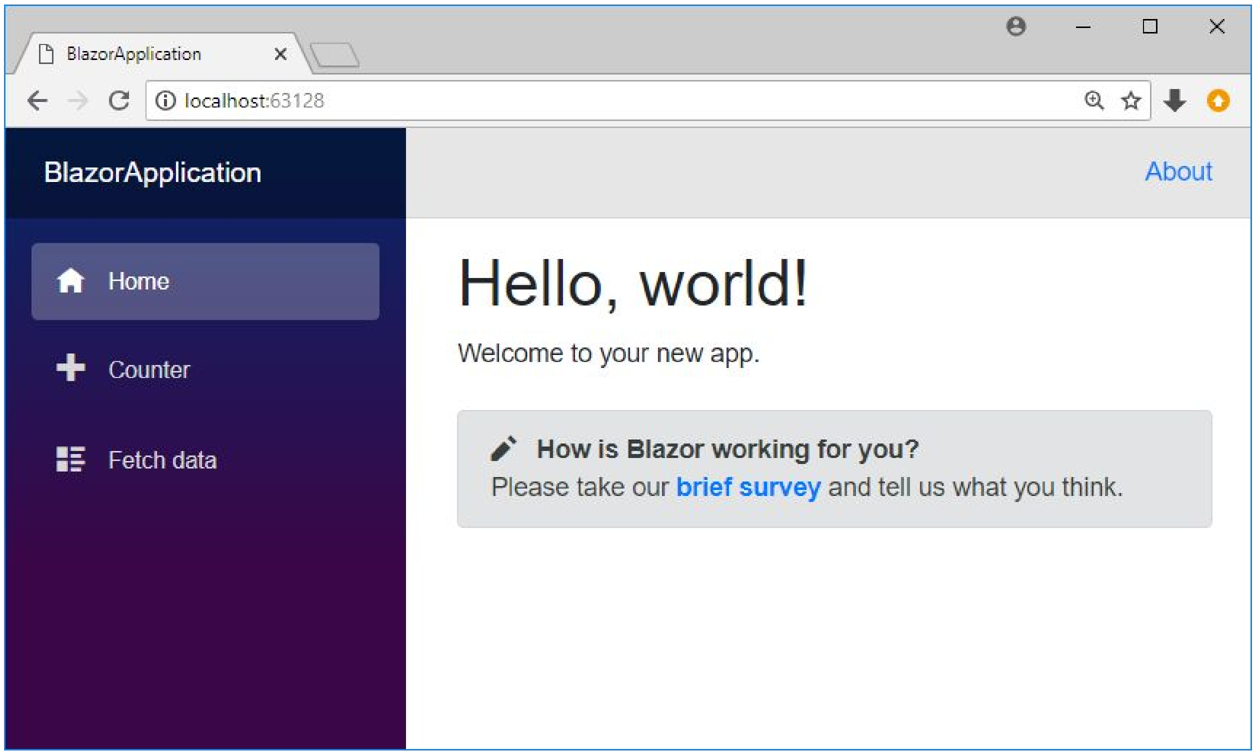
In the New Project dialog, select Web in the left pane and then choose ASP.NET Core Web Application in the middle pane. Specify the project name in the Name field and click OK.


New ASP.NET Core Web Application - BlazorApplication
?
.NET Core
ASP.NET Core 2.1
Learn more
Empty
API
Blazor
Blazor (ASP.NET Core hosted)
Blazor (Server-side in ASP.NET Core)
Web Application
Web Application (Model-View-Controller)
Angular
React.js
React.js and Redux
Razor Class Library
Learn more
Change Authentication
Enable Docker Support
OS:
Windows
Requires Docker for Windows
Docker support can also be enabled later Learn more
OK
Cancel

A project template for creating a standalone Blazor application that runs on WebAssembly.

Authentication **No Authentication**

On the New ASP.NET Core Web Application dialog, choose the Blazor template and select OK. Once the project is created, build and run the application.



Hello, world!

Welcome to your new app.

How is Blazor working for you?
Please take our **brief survey** and tell us what you think.

- The Blazor application runs in the browser, and it has three pages: Home, Counter, and Fetch data.
- These pages are implemented by the three Razor files in the Pages folder.
- Each of these files implements a Blazor component that is compiled and executed on the client-side in the browser.

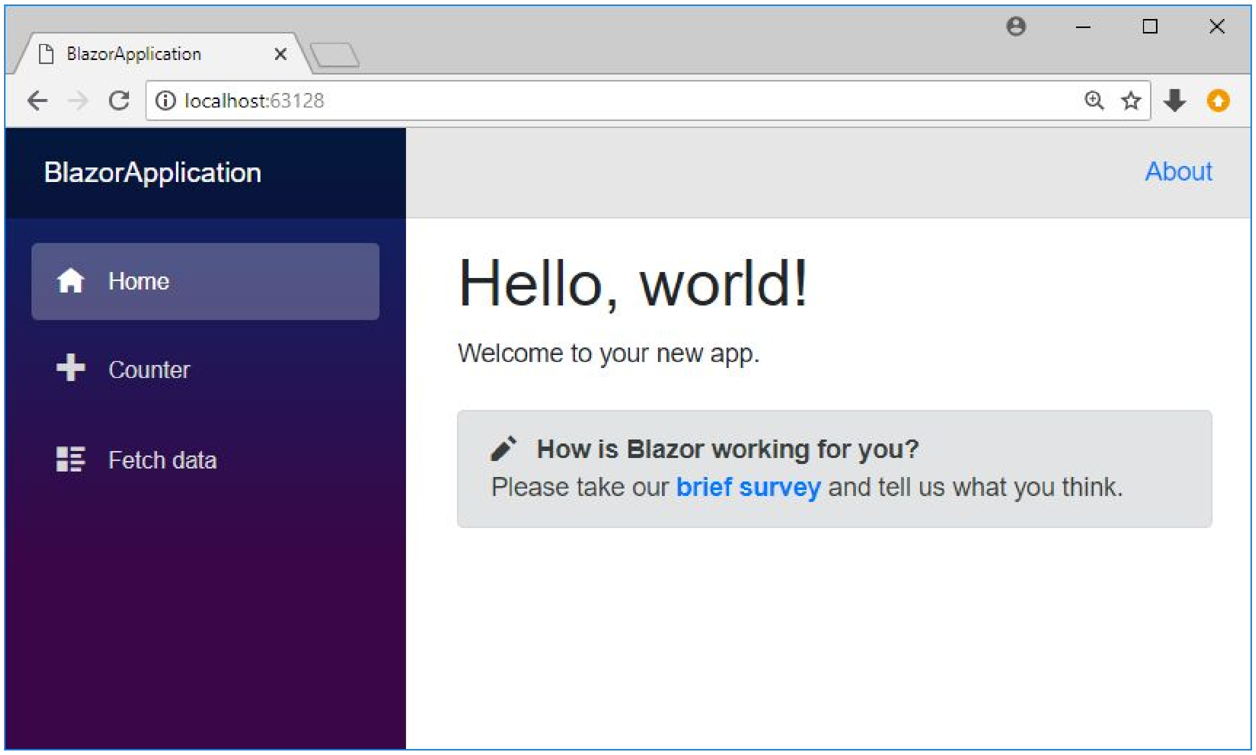
Home Page

Home page implementation is in Index.cshtml file.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Counter Page

Counter page has one button, and each time the button is selected, the counter is incremented without a page refresh.



Hello, world!

Welcome to your new app.

How is Blazor working for you?
Please take our **brief survey** and tell us what you think.

Counter page implementation is in Counter.cshtml file.

```
@page "/counter"

<h1>Counter</h1>

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

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

@functions {
    int currentCount = 0;

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

Normally, this kind of client-side behavior is handled in JavaScript; but here, it is implemented in C# and .NET by the Counter component.

Fetch Data

Take a look at the implementation of the FetchData component in FetchData.cshtml.

```
@page "/fetchdata"
@inject HttpClient Http

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from the server.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@functions {
    WeatherForecast[] forecasts;

    protected override async Task OnInitAsync()
    {
        forecasts = await
Http.GetJsonAsync<WeatherForecast[]>("sample-data/weather.json");
    }

    class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public int TemperatureF { get; set; }
        public string Summary { get; set; }
    }
}
```

The `@inject` directive is used to inject an HttpClient instance into the component. The FetchData component uses the injected HttpClient to retrieve JSON data from the server when the component is initialized.

Lab 2 Create Component from Code Behind

You can create view markup and C# code logic in separate files when creating a Blazor component.

- Using @inherits directive to tell the Blazor compiler to derive the class generated from the Razor view from class specified with this directive.
- The class specified with @inherits directive must be inherited from BlazorComponent class and it provides all base functionality for the component.

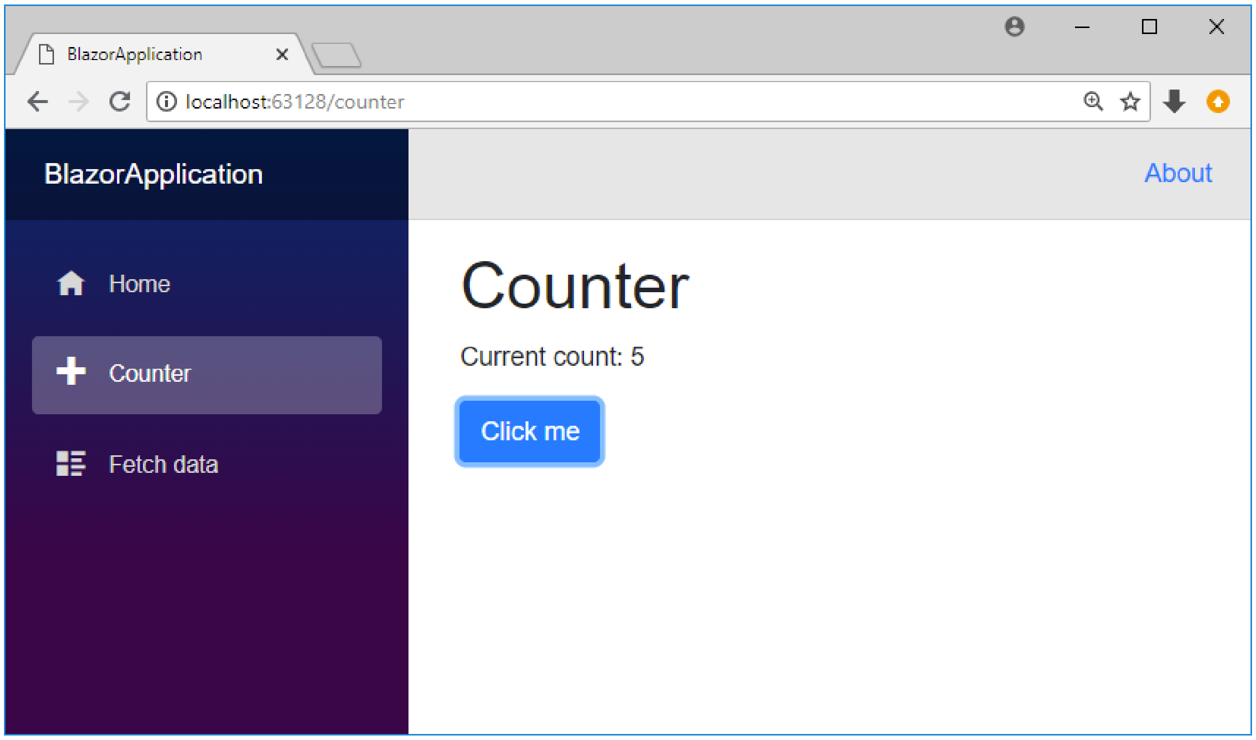
Let's move the C# code logic from Counter.cshtml file to seperate code behind class.

```
using Microsoft.AspNetCore.Blazor.Components;

namespace BlazorApplication
{
    public class CounterClass : BlazorComponent
    {
        public int CurrentCount { get; set; }

        public void IncrementCount()
        {
            CurrentCount += 5;
        }
    }
}
```

The Counter.cshtml file will use the properties and methods from the code behind class just by addingÂ `@inherits CounterClass`.

```
@page "/counter"
@inherits CounterClass

<h1>Counter</h1>

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

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

Blazor compiler generates class for all the view pages with the same class name as the page name, so here, a specified base class cannot have the same name as Razor View, otherwise, it would cause a compile-time error.



Now when the Click me button is pressed, the counter is incremented by 5.

Lab 3 Component Parameters

In Blazor, you can add parameters to any component which are defined using non-public properties on the component class by decorating that property with `[Parameter]` attribute.

```
public class CounterClass : BlazorComponent
{
    public int CurrentCount { get; set; }

    [Parameter]
    protected string SubTitle { get; set; }

    public void IncrementCount()
    {
        CurrentCount += 5;
    }
}
```

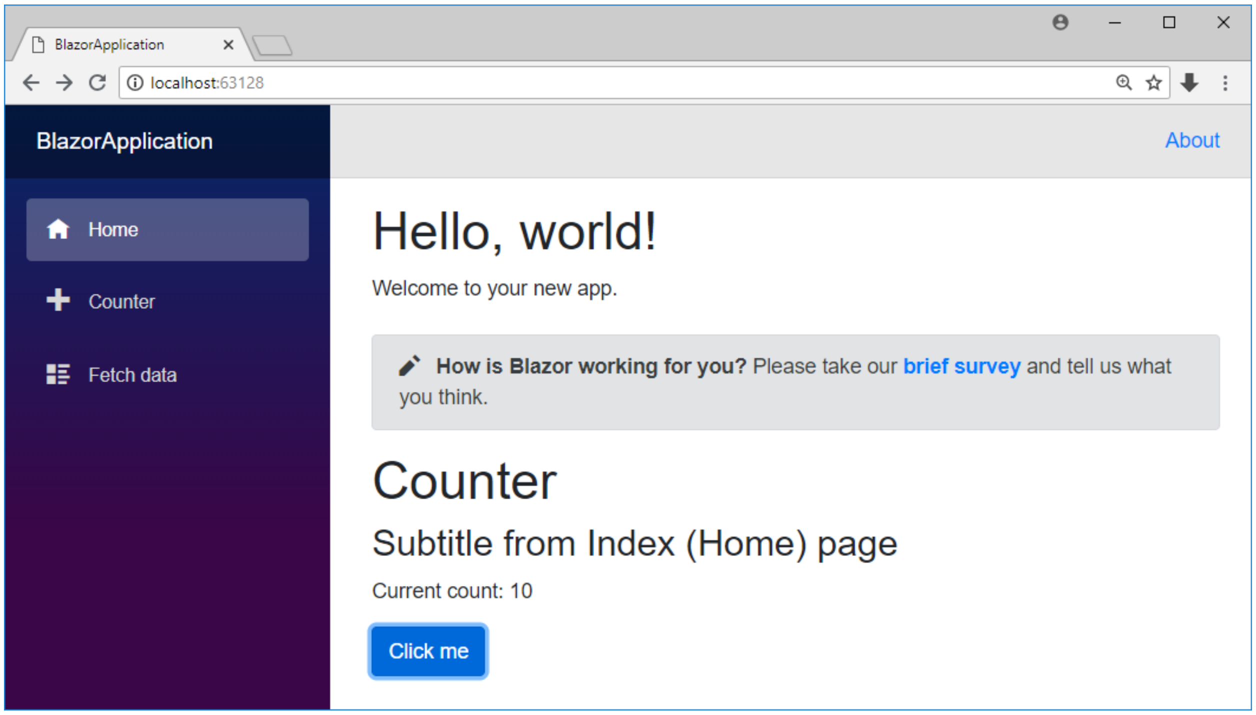
In markup, you can specify arguments (parameters) for a component using attributes. In the following example, the Home Component (Index.cshtml) sets the value of the SubTitle property of the Counter Component (Counter.cshtml):

```
<Counter SubTitle="Subtitle from Index (Home) page"/>
```

Let's add this markup to the Home (index.cshtml) component.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?"/>

<Counter SubTitle="Subtitle from Index (Home) page"/>
```

Now, you can see the subtitle of a counter component on a home page.



Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our **brief survey** and tell us what you think.

Counter

Subtitle from Index (Home) page

Current count: 10

Lab 4 Component Reusability

In Blazor, you can easily add and reuse any component in other components by declaring them using HTML element syntax. The markup for using a component looks like an HTML tag where the name of the tag is the component type.

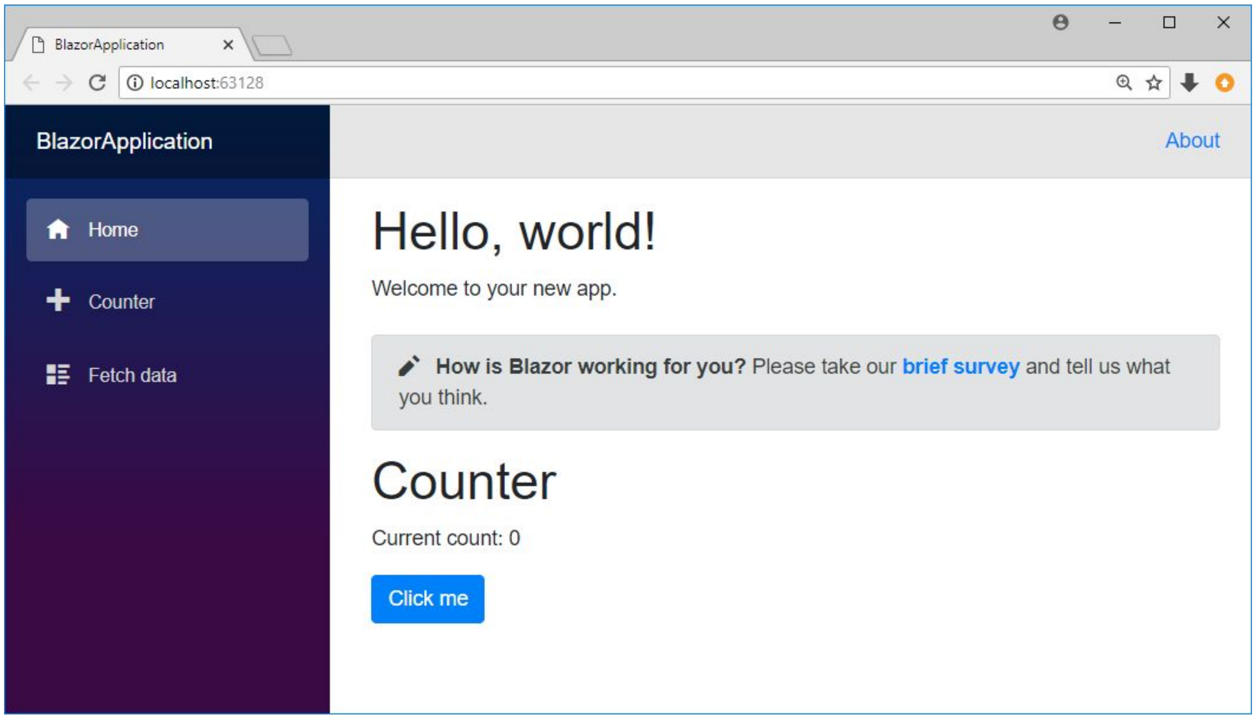
The following markup renders a Counter (counter.cshtml) instance.

```
<Counter />
```

Let's add this markup to the Home (index.cshtml) component.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

<Counter />
```

Now you can see the counter component on home page.



Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our **brief survey** and tell us what you think.

Counter

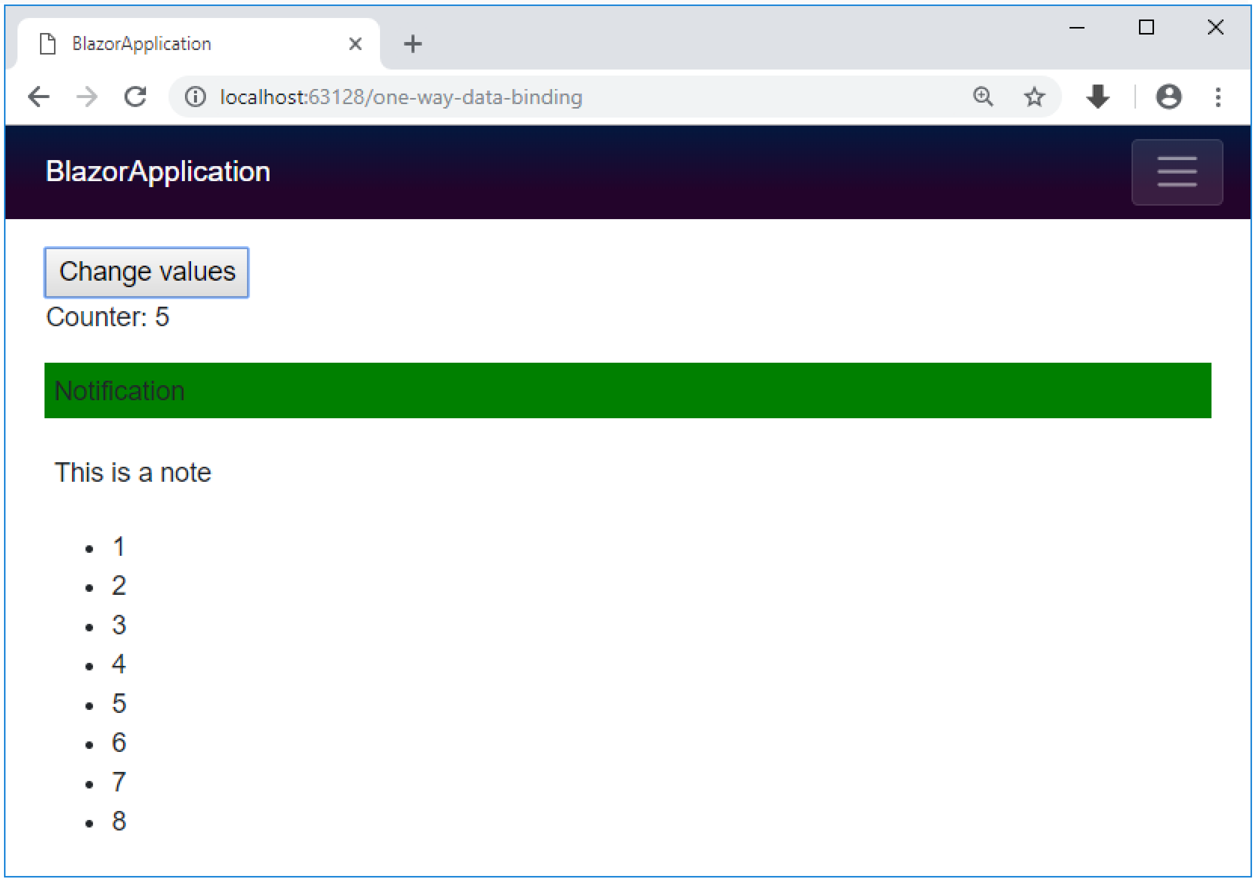
Current count: 0

Lab 5 One-way Data Binding

In other frameworks such as Angular, one-way data binding is also known as interpolation.

- In one-way binding, we need to pass property or variable name along with `@`, i.e., @Title (here Title is either the property or variable).
- In the following example, we have done one-way binding with different variables.

```
@page "/one-way-data-binding"

<!-- Use this button to trigger changes in the source values -->
<button onclick="@ChangeValues">Change values</button>

<p>Counter: @Count</p>

@if (ShowWarning)
{
    <p style="background-color: red; padding: 5px">Warning!</p>
}

<p style="background-color: @Background; color=white; padding:
5px">Notification</p>

<ul>
    @foreach (var number in Numbers)
    {
        <li>@number</li>
    }
</ul>

@functions {
    private int Count { get; set; } = 0;
    private bool ShowWarning { get; set; } = true;
    private string Background { get; set; } = "red";
    private List<int> Numbers { get; set; } = new List<int> { 1, 2, 3 };

    private void ChangeValues()
    {
        Count++;
        ShowWarning = !ShowWarning;
        Background = Background == "red" ? "green" : "red";
        Numbers.Add(Numbers.Max() + 1);
    }
}
```



Counter: 5

Notification

This is a note

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

Lab 6 Two-way Data Binding

Blazor also supports two-way data binding by using `bind` attribute. Currently, Blazor supports only the following data types for two-way data binding.

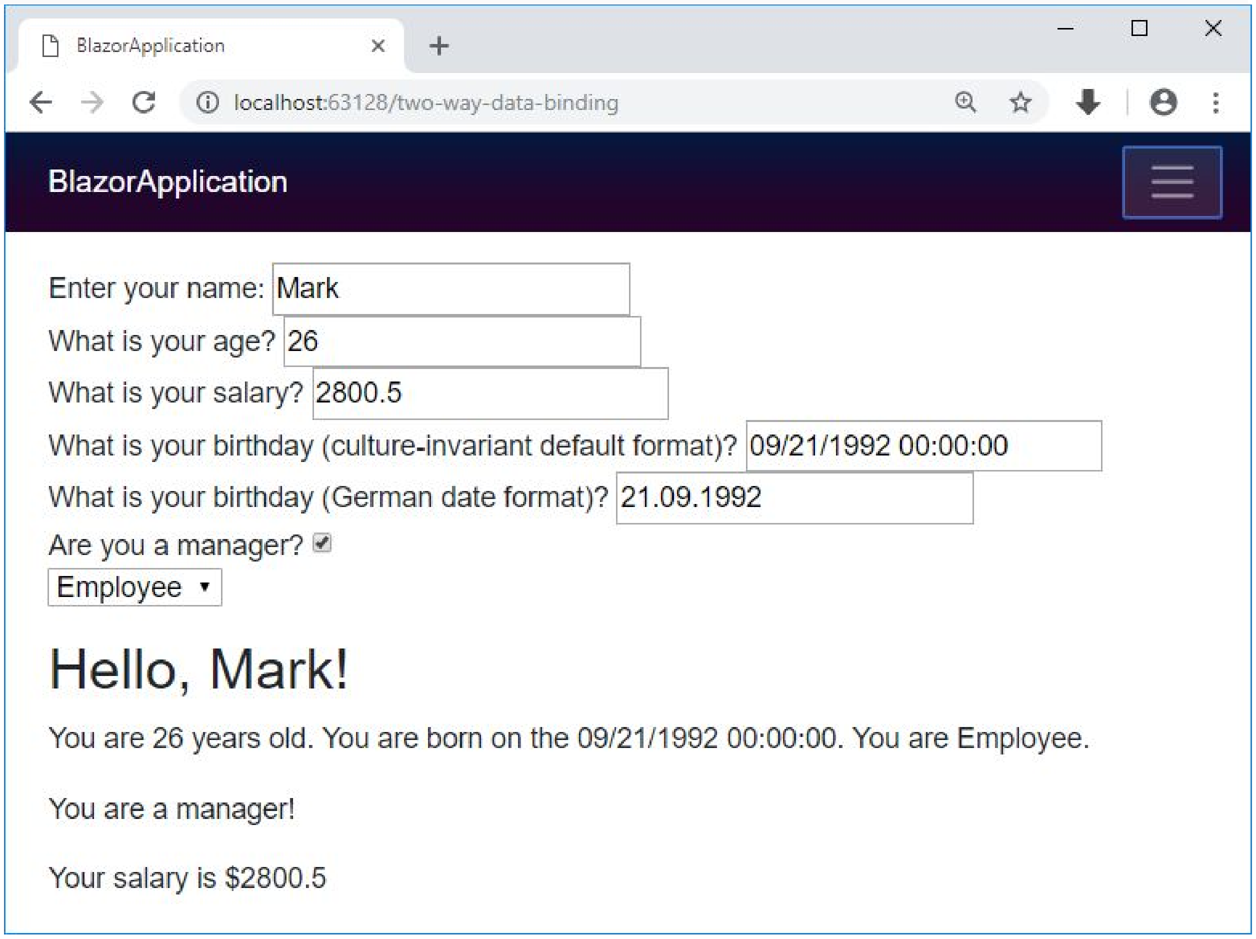
- string
- int
- DateTime
- Enum
- bool

If you need other types (e.g. decimal), you need to provide getter and setter from/to a supported type.

The following example demonstrates different two-way binding scenarios.

```
@page "/two-way-data-binding"

<p>
    Enter your name: <input type="text" bind="@Name" /><br />
    What is your age? <input type="number" bind="@Age" /><br />
    What is your salery? <input type="number" bind="@Salary" /><br />

    What is your birthday (culture-invariant default format)? <input
type="text" bind="@Birthday" /><br />
    What is your birthday (German date format)? <input type="text"
bind="@Birthday" format-value="dd.MM.yyyy" /><br />

    Are you a manager? <input type="checkbox" bind="@IsManager" /><br />

    <select id="select-box" bind="@TypeOfEmployee">
        <option
value=@EmployeeType.Employee>@EmployeeType.Employee.ToString()</option>
        <option
value=@EmployeeType.Contractor>@EmployeeType.Contractor.ToString()</option
>
        <option
value=@EmployeeType.Intern>@EmployeeType.Intern.ToString()</option>
    </select>
</p>

<h2>Hello, @Name!</h2>

<p>You are @Age years old. You are born on the @Birthday. You are
@TypeOfEmployee.</p>
```

```
@if (IsManager)
{
    <p>You are a manager!</p>
}

    <p>Your salary is $@Salary</p>

@functions {
private enum EmployeeType { Employee, Contractor, Intern };
private EmployeeType TypeOfEmployee { get; set; } = EmployeeType.Employee;

private string Name { get; set; } = "Mark";
private bool IsManager { get; set; } = true;
private static int Age { get; set; } = 26;
public DateTime Birthday { get; set; } = DateTime.Today.AddYears(-Age);

public decimal Salary { get; set; } = (decimal) 2800.5;
}
```



Enter your name: Mark

What is your age? 26

What is your salary? 2800.5

What is your birthday (culture-invariant default format)? 09/21/1992 00:00:00

What is your birthday (German date format)? 21.09.1992

Are you a manager? ✓

Employee ▾

Hello, Mark!

You are 26 years old. You are born on the 09/21/1992 00:00:00. You are Employee.

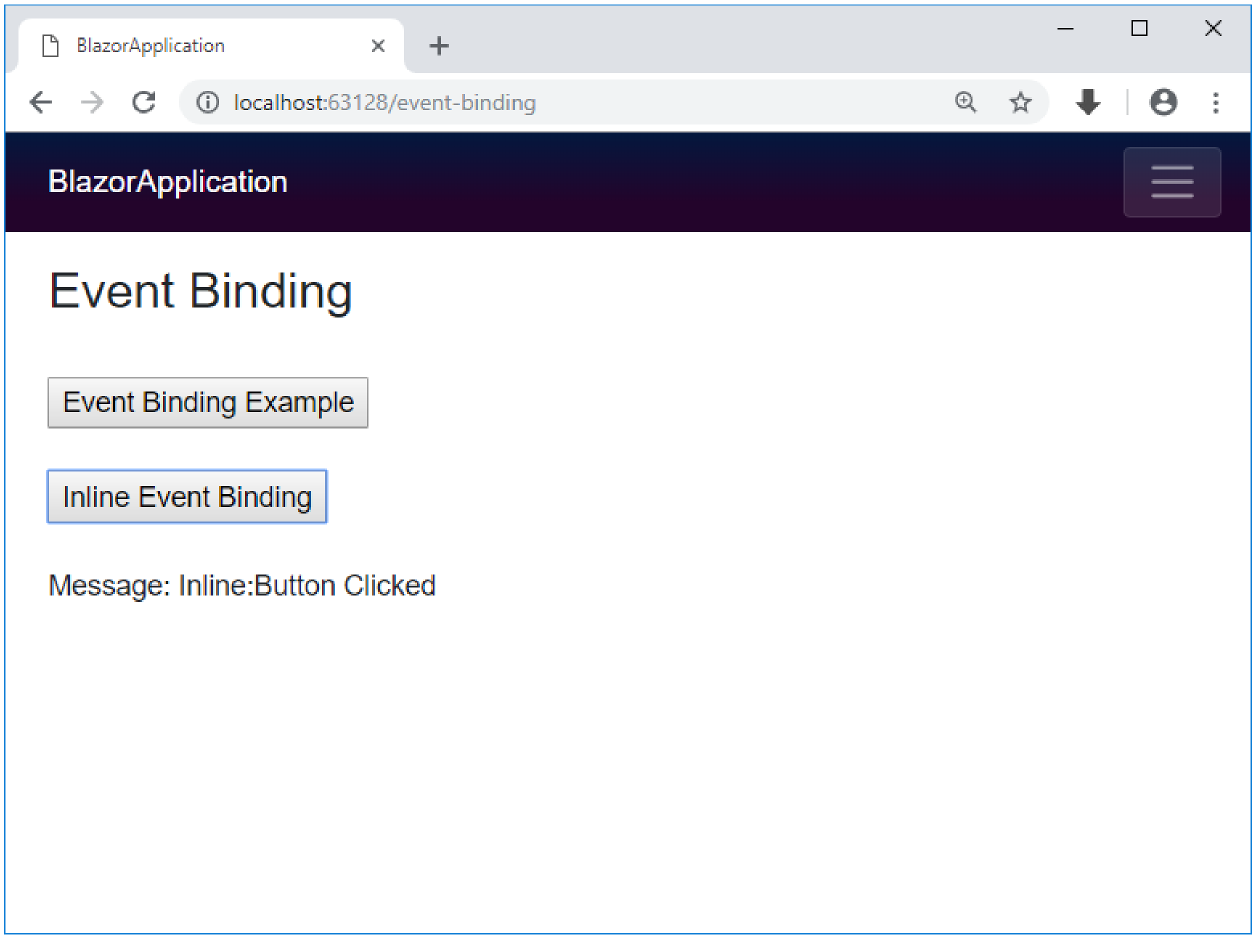
You are a manager!

Your salary is $2800.5

Lab 7 Event Binding

Event binding is quite limited in Blazor. Currently, only `onclick` and `onchange` are supported. You can find more details in the Blazor GitHub.

```
@page "/event-binding"

<h3>Event Binding</h3> <br /> <br />
<button onclick=@ButtonClicked>Event Binding Example</button>  <br /> <br
/>
<button onclick=@(() => Message = "Inline:Button Clicked")>Inline Event
Binding</button> <br /> <br />
<p>Message: @Message</p>

@functions {
    private string Message { get; set; } = "";
    void ButtonClicked()
    {
        Message = "Button Clicked";
    }
}
```



Event Binding

Message: Inline:Button Clicked

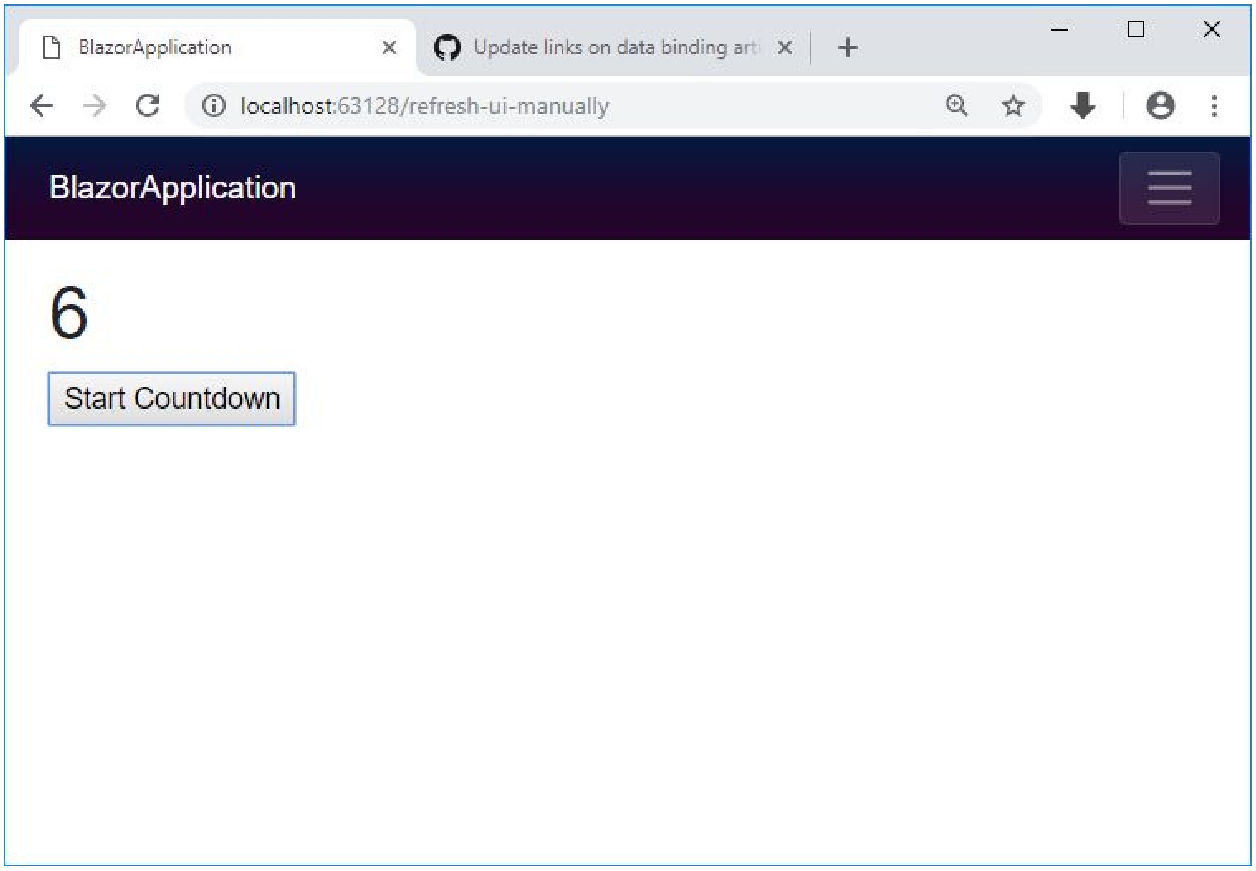
Lab 8 Refresh UI Manually

By default, Blazor detects a necessary UI refresh automatically in many scenarios like button click etc. However, there are situations in which you want to trigger a UI refresh manually by using the `BlazorComponent.StateHasChanged` method.

In the following sample, it changes the application's state using a timer.

```
@page "/refresh-ui-manually"
@using System.Threading;

<h1>@Count</h1>

<button onclick=@StartCountdown>Start Countdown</button>

@functions {
    private int Count { get; set; } = 10;

    void StartCountdown()
    {
        var timer = new Timer(new TimerCallback(_ =>
        {
            if (Count <= 0) return;
            Count--;

            // Note that the following line is necessary because otherwise
            // Blazor would not recognize the state change and not refresh
the UI
            this.StateHasChanged();
        }), null, 1000, 1000);
    }
}
```

It will start the count down when you click on Start Countdown button.


BlazorApplication
Update links on data binding arti
localhost:63128/refresh-ui-manually
BlazorApplication
6
Start Countdown

Lab 9 Layout

Typically, modern web applications contain more than one page with certain layout elements such as menus, logos, copyright messages, etc. which are present on all pages.

- Reusing the code of these layout elements onto all pages would not be a good solution.
- It becomes hard to maintain and probably inconsistent over time.
- Blazor Layouts solve this problem. Every part of the application in Blazor is a component, so Layout is also considered as a component in Blazor applications.

Blazor Layout

A Blazor layout is just another Blazor component which is also defined in a Razor template or in C# code, it can contain everything just like Blazor component such as data binding, dependency injection is supported, etc. but Blazor layout includes two additional aspects:

Inherit from BlazorLayoutComponent

- The layout component inherits from `BlazorLayoutComponent`.
- It adds a property Body to the component which contains the content to be rendered inside the layout.

Body Keyword

- The layout component contains the Razor keyword `@Body`.
- During rendering, it is replaced by the content of the layout.

The default template of Blazor contains MainLayout.cshtml under the shared folder. It works as Layout page.

```
@inherits BlazorLayoutComponent

<div class="sidebar">
    <NavMenu />
</div>

<div class="main">
    <div class="top-row px-4">
```

```
        <a href="http://blazor.net" target="_blank"
class="ml-md-auto">About</a>
    </div>

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

The above code sample shows the Razor template of a layout component.

Load Blazor Component

To load any Blazor component into a layout, you will need to use `@layout` directive.

```
@layout MyLayout

@page "/my-data"

<h2>My Data</h2>
...
```

- When the component is requested its content is loaded into the layout component at the point the `@Body` tag is defined.
- The compiler converts this directive into a LayoutAttribute, which is applied to the component class.

If you are writing a component in pure C# and want to to use a layout component. Then you can decorate it with the [LayoutAttribute]

```
// Using pure C#

[Layout(typeof(MyLayout))]
public class MyClass : BlazorComponent
{
    ...
}
```

This is ultimately what the compiler will convert the @layout directive to anyway.

Global/Centralized Layout

In ASP.net MVC, the `_ViewImports.cshtml` file was introduced to provide a mechanism to make directives available to Razor pages globally without adding them to each page.

- You can also use this file to define Layout page in Blazor.
- Layout is automatically applied to all the Blazor pages in the folder hierarchy if you Layout page in this file.

The default `_ViewImports.cshtml` file looks like this when a new Blazor application is created.

```
@layout MainLayout
```

Nested Layout

In Blazor, you can also use nested layouts; i.e., Blazor component can reference a layout which references another layout.

- Nesting layouts can be used to reflect a multi-level menu structure.

Here is the `MyLayout.cshtml`.

```
@inherits BlazorLayoutComponent

<div class="main">
    <div class="top-row px-4">
        <h3>My Layout</h3>
    </div>

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

Here is the `CustomLayout.cshtml`, which is referencing `MyLayout.cshtml`.

```
@layout MyLayout
@inherits BlazorLayoutComponent
```

```
<div class="main">
    <div class="row">
        <div class="col-sm-12">
            <h2>Nested Layout Example</h2>
        </div>
    </div>
    <div class="row">
        <div class="col-sm-12">
            @Body
        </div>
    </div>
</div>
```

Here is the Nestedlayout.cshtml which is referencing `CustomLayout.cshtml`.

```
@layout CustomLayout
@page "/nestedlayout"

<h3>@Title</h3>

@functions {
string Title = "This is child page using Nested Layout";
}
```

Layout page reduces duplicate code in your application and make the look and feel consistent throughout the application.

Lab 10 Routing

What are the Route and Routing?

A Route is a URL pattern, and Routing is a pattern matching process that monitors the requests and determines what to do with each request.

- Blazor provides a client-side router.
- The Microsoft.AspNetCore.Blazor.Routing.Router class provides Routing in Blazor.
- In Blazor, the `<Router>` component enables routing, and a route template is provided to each accessible component.
- The `<Router>` component appears in the `App.cshtml` file.

```
<!--
    Configuring this here is temporary. Later we'll move the app config
    into Program.cs, and it won't be necessary to specify AppAssembly.
-->
<Router AppAssembly=typeof(Program).Assembly />
```

Route Templates

In Blazor, you define routes using route templates. You can define a route template by adding the `@page` directive to the top of a component.

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.
```

The above component would be loaded when the user navigated to www.mydomaim.com/

Multiple Routes

It is also valid to specify multiple route templates for a component. You can achieve this by defining multiple `@page` directives.

```
@page "/"
@page "/index"

<h1>Hello, world!</h1>

Welcome to your new app.
```

The above component would be loaded when the user navigated to www.mydomaim.com/ or www.mydomaim.com/index.

Define Route for pure C# Component

If you are defining your component as a pure C# class, then you can specify its route template by decorating it with the route attribute.

```
[Route("/counter")]
public class CounterClass : BlazorComponent
{
    // code here
}
```

It is ultimately what the `@page` directive gets compiled to.

- When a *.cshtml file with an @page directive is compiled, the generated class is given a RouteAttribute specifying the route template.
- At runtime, the router looks for component classes with a RouteAttribute and renders whichever component has a route template that matches the requested URL.

Route parameters

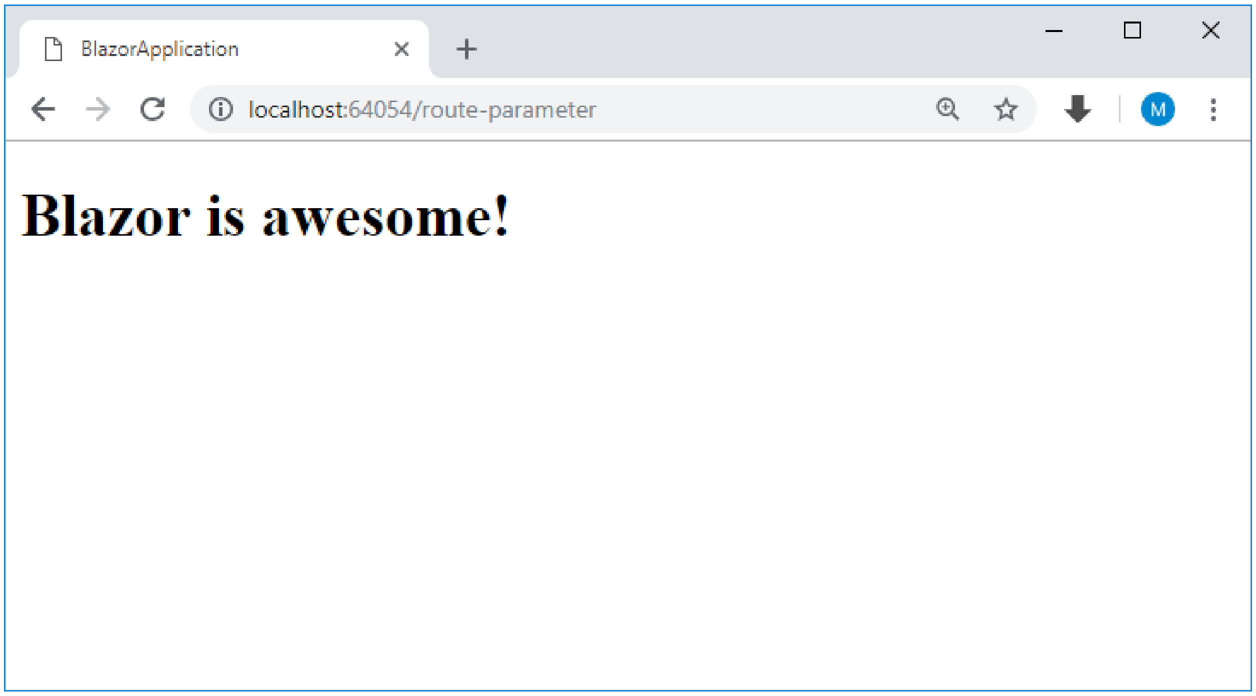
The Blazor client-side router uses route parameters to populate the corresponding component parameters with the same name (case insensitive).

```
@page "/route-parameter"
@page "/route-parameter/{text}"

<h1>Blazor is @Text!</h1>
```

```
@functions {
    [Parameter]
    private string Text { get; set; } = "awesome";
}
```

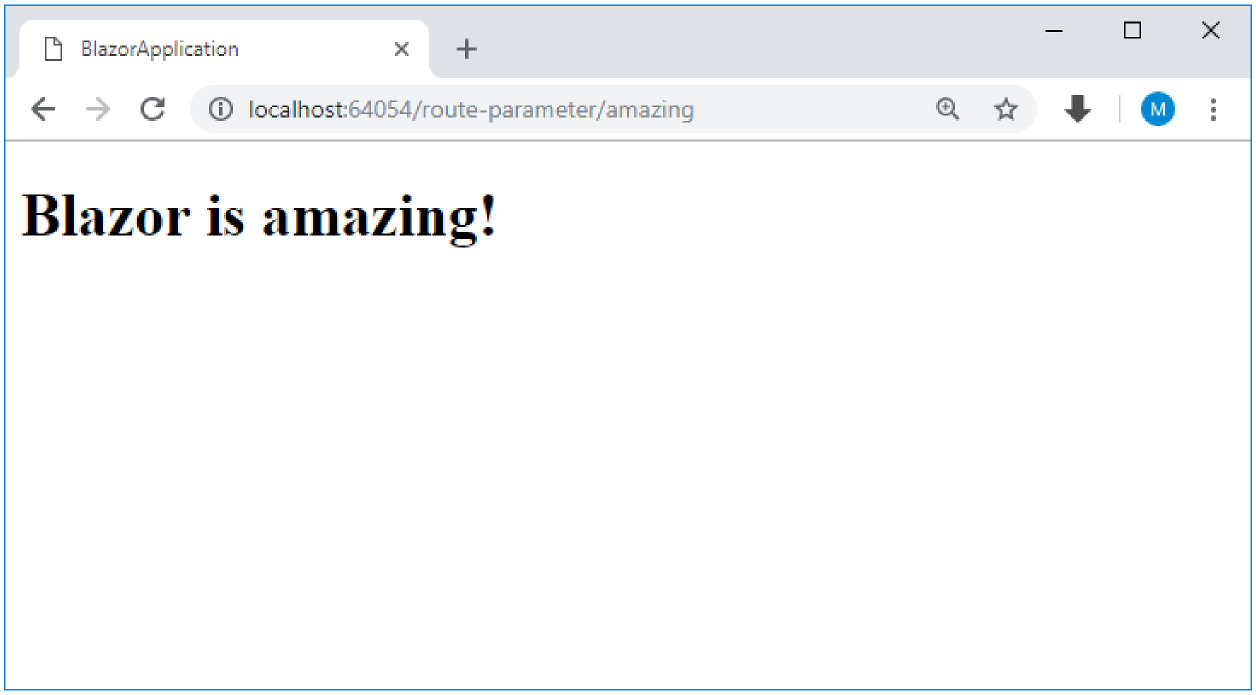
Optional parameters aren't supported yet, so two `@page` directives are applied in the above example.

The first `@page` directive permits navigation to the component without a parameter.



Blazor is awesome!

The second `@page` directive takes the `{text}` route parameter and assigns the value to the Text property.



Blazor is amazing!

Lab 11 Dependency Injection

Dependency Injection (DI) is a technique for achieving Inversion of Control (IoC) between classes and their dependencies. In other words, it is a technique for accessing services configured in a central location.

- Blazor has built-in support for dependency injection (DI).
- Blazor applications can use built-in services by having them injected into components.
- Blazor apps can also define custom services and make them available via DI.

DI Service Configuration

Blazor's DI system is based on the DI system in ASP.NET Core. DI services can be configured with the following lifetimes:

Method	Description
Singleton	DI creates a single instance of the service. All components requiring this service receive a reference to this instance.
Transient	Whenever a component requires this service, it receives a new instance of the service.
Scoped	Blazor doesn't currently have the concept of DI scopes. Scoped behaves like Singleton. Therefore, prefer Singleton and avoid Scoped.

Default services

By default, Blazor's `BrowserServiceProvider` automatically adds the following services to the service collection of an application.

Method	Description

IUriHelper	Helpers for working with URIs and navigation state (singleton).
HttpClient	Provides methods for sending HTTP requests and receiving HTTP responses from a resource identified by a URI (singleton).

Add services to DI

Here is a simple service which will retrieve employee's data asynchronously.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace BlazorApplication.Services
{
    public class Employee
    {
        public string FirstName { get; set; }

        public string LastName { get; set; }
    }

    public interface IRepository
    {
        Task<IReadOnlyList<Employee>> GetAllEmployeesAsync();
    }

    public class Repository : IRepository
    {
        private static Employee[] Employees { get; set; } = new[]
        {
            new Employee { FirstName = "Andy", LastName = "White" },
            new Employee { FirstName = "Mark", LastName = "Doe" }
        };

        public async Task<IReadOnlyList<Employee>> GetAllEmployeesAsync()
        {
            await Task.Delay(100);

            return Repository.Employees;
        }
    }
}
```

When a new Blazor application is created, you will see `Startup.cs` file which contains `ConfigureServices` and `Configure` methods.

```
using Microsoft.AspNetCore.Blazor.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace BlazorApplication
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
        }

        public void Configure(IBlazorApplicationBuilder app)
        {
            app.AddComponent<App>("app");
        }
    }
}
```

The ConfigureServices method has IServiceCollection as an argument, which is a list of service descriptor objects (ServiceDescriptor). You can add services to the service collection by providing service descriptors.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IRepository, Repository>();
}
```

Injecting Services

To inject a service in a component, use the `@inject` keyword as shown in the following sample.

```
@using BlazorApplication.Services
@page "/"
```

```
@inject IRepository Repository

<h1>Hello, world!</h1>
<ul>
    @if (Employees != null)
    {
        @foreach (var emp in Employees)
         {
             <li>@emp.FirstName @emp.LastName</li>
         }
    }
</ul>

@functions {
    private IReadOnlyList<Employee> Employees;

    protected override async Task OnInitAsync()
    {
        Employees = await Repository.GetAllEmployeesAsync();
    }
}
```

Technically, it generates a property and decorate it with the `InjectAttribute` attribute.

If a base class is required for Blazor components and injected properties are also required for the base class, then add `InjectAttribute` manually.

```
public class ComponentBase : BlazorComponent
{
    [Inject]
    protected IRepository Repository { get; set; }
    ...
}
```

Lab 12 Lifecycle Methods

The Blazor application provides different synchronous as well as asynchronous lifecycle methods.

- OnInitialized
- OnInitializedAsync
- OnParametersSet
- OnParametersSetAsync
- OnAfterRender
- OnAfterRenderAsync
- ShouldRender

OnInit & OnInitAsync

The synchronous and asynchronous version of the application methods which gets executed when the component gets Initialized.

- The `OnInitialized` is called first, then `OnInitializedAsync`.
- It is executed when the component is completely loaded.
- You can use this method to load data from services because each control in the UI is loaded after this method.
- It is executed when the component is ready and when it has received the values from the parent in the render tree.
- Any asynchronous operations, which require the component to re-render once they complete, should be placed in the `OnInitializedAsync` method.

```
@page "/"

<h1>OnInit & OnInitAsync Demo</h1>

@foreach (var item in EventType)
{
    @item <hr />
}

@functions{
    List<string> EventType = new List<string>();

    protected override void OnInitialized()
    {
        EventType.Add("OnInitialized is called");
    }
    protected override async Task OnInitializedAsync()
    {
        EventType.Add("OnInitializedAsync is called");
        await Task.Delay(1000);
    }
}
```

OnParametersSet & OnParametersSetAsync

The synchronous and asynchronous way of setting the parameter when the component receives the parameter from its parent component.

- The `OnParametersSet` and `OnParametersSetAsync` methods are called when a component is first initialised.
- After initialisation, `OnParametersSet` and `OnParametersSetAsync` are called each time new or updated parameters are receieved from the parent in the render tree.

```
@page "/"

<h1>OnParametersSet & OnParametersSetAsync Demo</h1>

@foreach (var item in EventType)
{
   @item <hr />
}

@functions{
   List<string> EventType = new List<string>();

   protected override void OnParametersSet()
   {
       EventType.Add("OnParameterSet is called");
   }
   protected override async Task OnParametersSetAsync()
   {
       EventType.Add("OnParametersSetAsync is called");
       await Task.Delay(1000);
   }
}
```

OnAfterRender & OnAfterRenderAsync

The synchronous and asynchronous version of the application methods to perform the additional steps like initializing the other components.

- The `OnAfterRender` and `OnAfterRenderAsync` methods are called after each render of the component.
- At the point they are called you can expect that all element and component references are populated.
- It means that if you need to perform an action, such as attaching an event listener, it requires the elements of the component to be rendered in the DOM.
- Another great use for these lifecycle methods is for JavaScript library initialization, which requires DOM elements to be in place to work.

```
@page "/"

<h1>OnInit & OnInitAsync Demo</h1>

@foreach (var item in EventType)
{
   @item <hr />
}

@functions{
   List<string> EventType = new List<string>();

   protected override void OnAfterRender()
   {
       EventType.Add("OnAfterRender is called");
   }

   protected override  async Task OnAfterRenderAsync()
   {
       EventType.Add("OnAfterRenderAsync is called");
       await Task.Delay(1000);
   }
}
```

ShouldRender

This method returns a boolean value, if returns true, it means refresh the UI, otherwise changes are not sent to UI. The `ShouldRender` method always does the initial rendering despite its return value.

```
@page "/"

 <h1>ShouldRender Method Demo</h1>
@foreach (var item in EventType){  @item <hr />}
 @functions{
    List<string> EventType = new List<string>();

    protected override bool ShouldRender()
    {
        EventType.Add("ShouldRender is called");
        return true;
    }
}
```

StateHasChanged

This method notifies the component that its state has changed.

- It is called after any lifecycle method has been called.
- It can also be invoked manually to trigger a UI re-render.
- This method looks at the value returned from ShouldRender to decide if a UI re-render should happen.
- However, this only happens after the component has been rendered for the first time.

```
@page "/refresh-ui-manually"
@using System.Threading;

<h1>@Count</h1>

<button onclick=@StartCountdown>Start Countdown</button>

@functions {
    private int Count { get; set; } = 10;
    void StartCountdown()
    {
        var timer = new Timer(new TimerCallback(_ =>
        {
            if (Count <= 0) return;
            Count--;

            // Note that the following line is necessary because otherwise
            // Blazor would not recognize the state change and not refresh
the UI
            this.StateHasChanged();
        }), null, 1000, 1000);
    }

}
```

Lab 13 Invoke JavaScript Functions

Blazor is a frontend framework, but it has no direct access to the browser's DOM API.

- As web developers, we want to access browser APIs and existing JavaScript functions.
- There are times when Blazor .NET code is required to call a JavaScript function.
- For example, a JavaScript call can expose browser capabilities or functionality from a JavaScript library to the Blazor app.

JavaScript Interop

A Blazor app can invoke JavaScript functions from .NET and .NET methods from JavaScript code. This property of calling a JS method from C# code and vice versa is referred to as JavaScript Interop.

- Blazor uses JavaScript Interop to handle DOM manipulation and browser API calls.
- To call a JavaScript from .NET, use the `IJSRuntime` abstraction, which is accessible from `JSRuntime.Current`.

Let's add two JavaScript functions to `index.html` file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width">
    <title>BlazorApplication</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>Loading...</app>

    <script src="_framework/blazor.webassembly.js"></script>
    <script type="blazor-boot">
    </script>
    <script>
       function JSMethod() {
            $("#demop").text("JavaScript function called from C#");
        }
    </script>
    <script>
        function CSMethod() {
            DotNet.invokeMethodAsync('BlazorApplication',
'CSCallBackMethod');
        }
    </script>
</body>
</html>
```

- The `JSMethod` function will set the text of a `<p>` tag having id "demop" to "JavaScript function called from C#".
- This `CSMethod` function will have a call back to our C# method `CSCallBackMethod` which we will define later.

Now create a new Blazor component `JSInterop.cshtml` and add the following functions.

```
@functions {
    protected static string message { get; set; }

    protected void CallJSMethod()
    {
        JSRuntime.Current.InvokeAsync<bool>("JSMethod");
    }

    protected void CallCSMethod()
    {
        JSRuntime.Current.InvokeAsync<bool>("CSMethod");
    }

    [JSInvokable]
    public static void CSCallBackMethod()
    {
        message = "C# function called from JavaScript";
    }
}
```

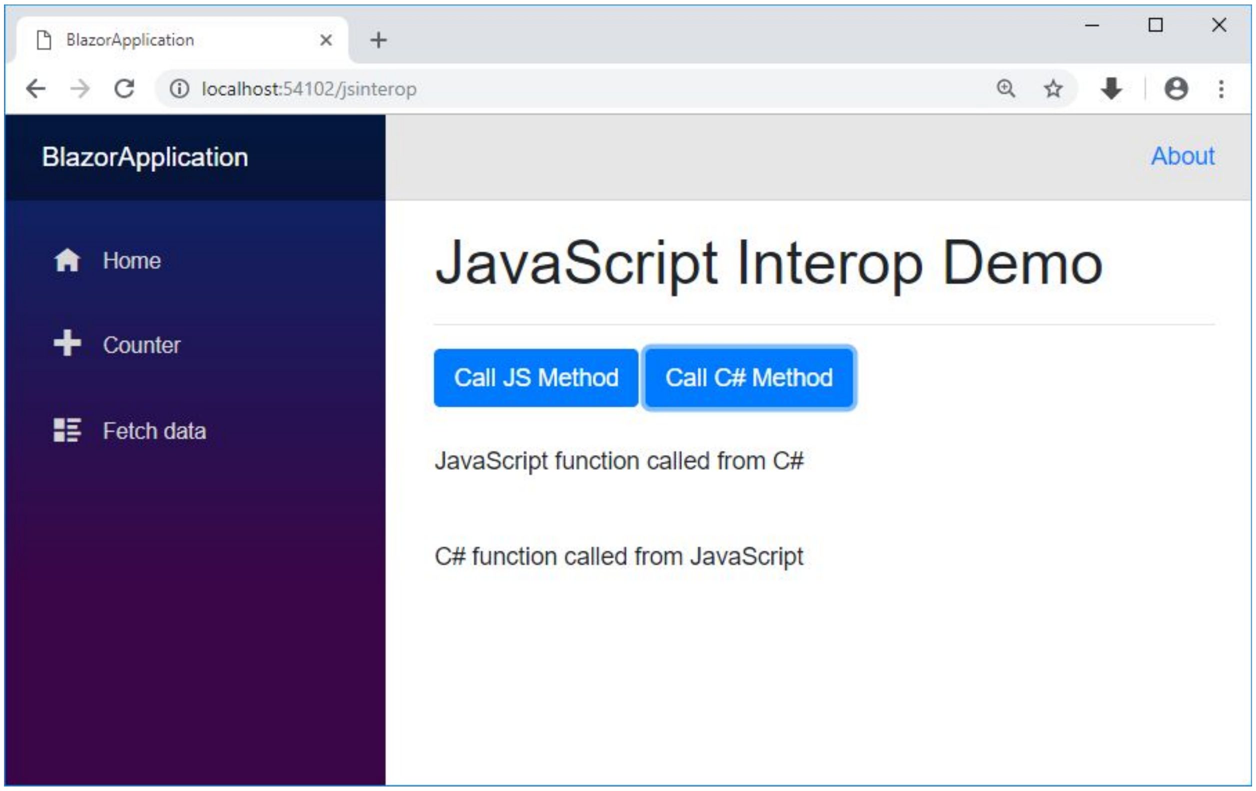
- The `CallJSMethod` and `CallCSMethod` will call our JS functions `JSMethod` and `CSMethod` by using `JSRuntime.Current.InvokeAsync` method.
- The `JSRuntime.Current.InvokeAsync` method can take two parameters; the JS function name and any parameter that needed to be supplied to the JS function. But, we are not passing any parameter to JS function.
- The `CSCallBackMethod` is a static method, and it will be called from the JavaScript function `CSMethod`.

- It is decorated with `[JSInvokable]` attribute.
- This method will set the value of a string variable message, which will be displayed on the UI.

Add two buttons to the `JSInterop.cshtml` file will call the `CallJSMethod` and `CallCSMethod` method on the `onclick` event.

```
@page "/jsinterop"
@using BlazorApplication.Pages
@using Microsoft.JSInterop

<h1>JavaScript Interop Demo</h1>

<hr />

<button class="btn btn-primary" onclick="@CallJSMethod">Call JS
Method</button>
<button class="btn btn-primary" onclick="@CallCSMethod">Call C#
Method</button>
<br />
<p id="demop"></p>
<br />
<p>@message</p>

@functions {
    protected static string message { get; set; }

    protected void CallCSMethod()
    {
        JSRuntime.Current.InvokeAsync<bool>("CSMethod");
    }

    protected void CallJSMethod()
    {
        JSRuntime.Current.InvokeAsync<bool>("JSMethod");
    }

    [JSInvokable]
    public static void CSCallBackMethod()
    {
        message = "C# function called from JavaScript";
    }
}
```

Click on the buttons to call the JavaScript functions and C# method.



JavaScript Interop Demo

JavaScript function called from C#

C# function called from JavaScript