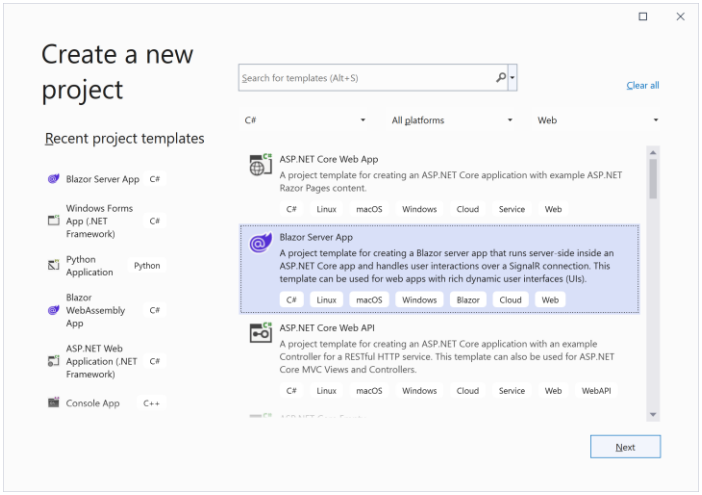
CPSC 555 – Assignment #3

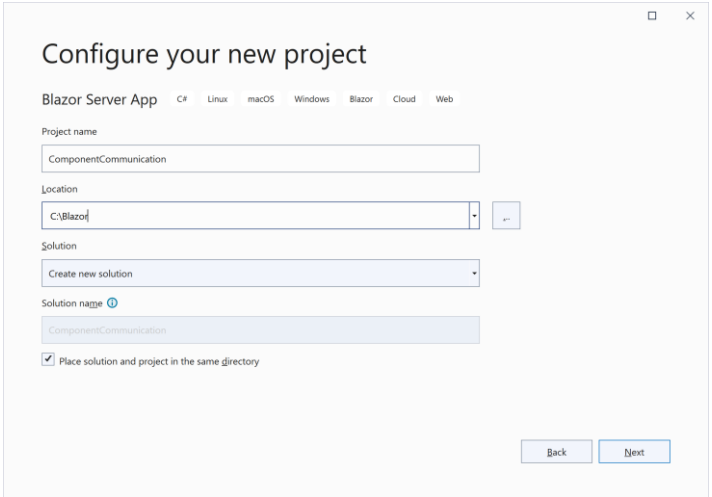
Component Communication

The purpose of this assignment is to understand the data communication issues between the pages/components of Blazor server-side web applications. There are different needs where data needs to be passed from one component to another. These include:

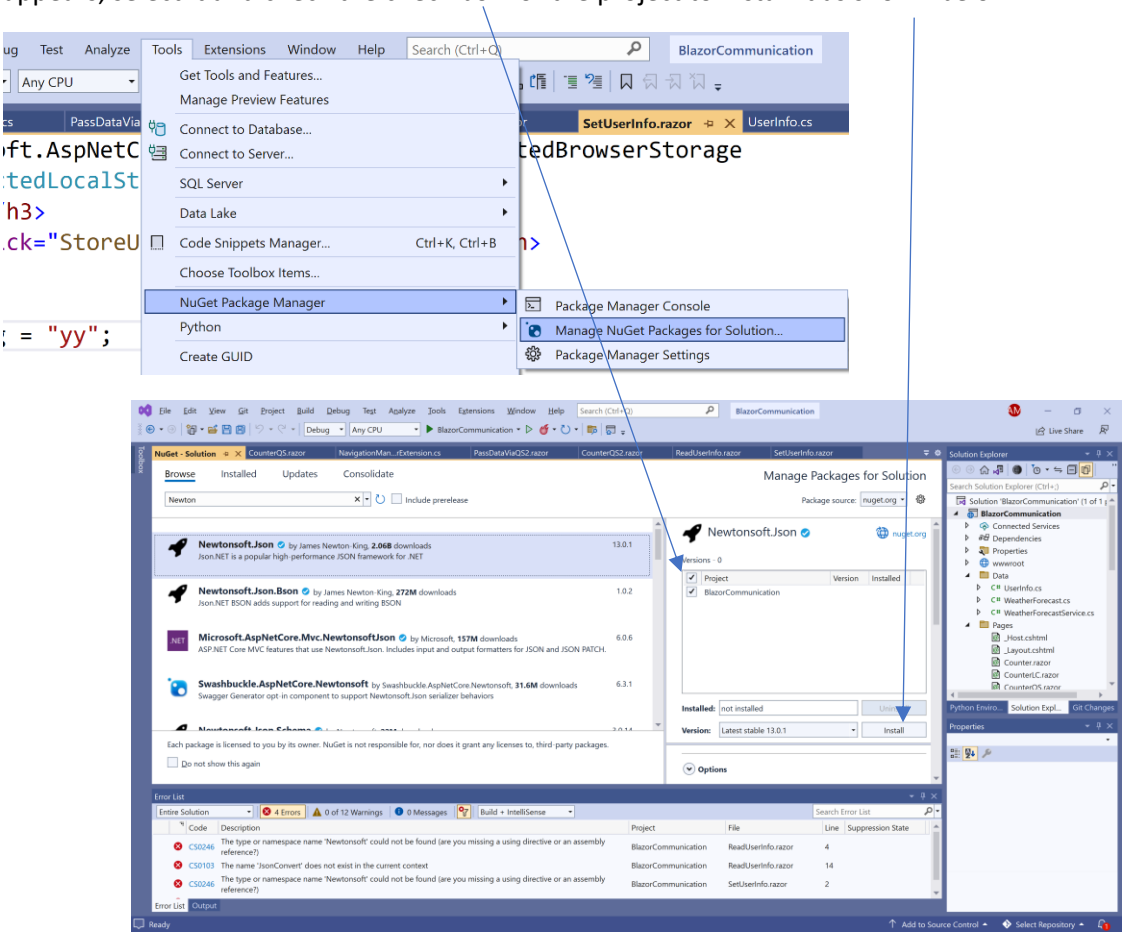
- Parent to child component data communication.
- Child to parent component data communication and Event callbacks.
- URL to component communication via Route Parameters and Query Strings.
- Component to component communication using Local and Session Storage.
- Page to Page communication via ProtectedSessionStore and ProtectedLocalStore

The above techniques were explained in lectures. You will create a Blazor server app type of project called ComponentCommunication.





Note that for the local and session storage, you will need to install Nuget package called NewtonSoft.json. From the Tools menu, select “Nuget Package Manager -> Manage Nuget Packages for Solution”, then click on the browse link and type Newton in the search box. Once NewtonSoft.json appears, select it and check the check box for the project to install it as shown below.



Initial Data from Parent to Child:

Modify the code in the Counter.razor to appear as:

```
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

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

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

@code {
    [Parameter]
    public int InitialCount { get; set; } = 0;
    [Parameter]
    public int IncBy { get; set; } = 1;
```

```
    private int currentCount = 0;

    protected override void OnInitialized()
    {
        currentCount = InitialCount;
    }

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

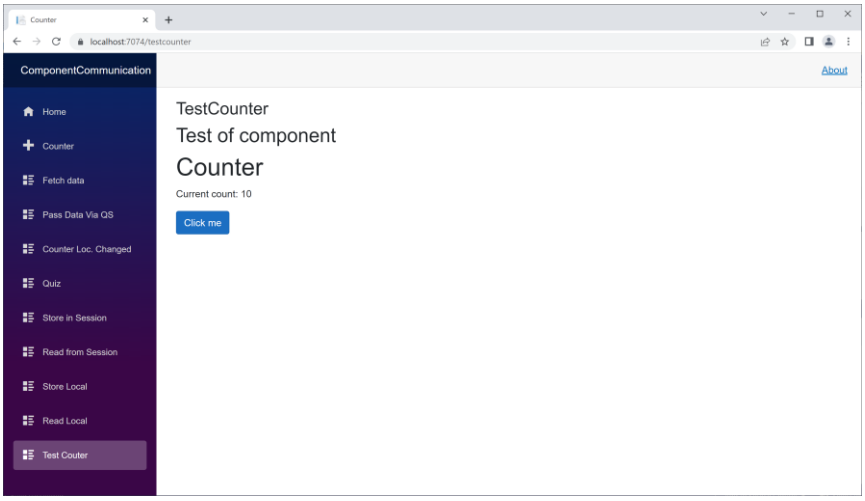
We have created two parameters called InitialCount and IncBy that the user of this component can provide at the time of creation. Add a razor component called TestCounter.razor to the Pages folder with the following code in it. It invokes the Counter and sets the InitialCount to 10 and IncBy to 5.

```
@page "/testcounter"
<h3>TestCounter</h3>
<h2>Test of component</h2>
<Counter InitialCount="10" IncBy="5"/>
@code {

}
```

Add a link to the NavMenu.razor for TestCounter page as:

```
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="testcounter">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Test Couter
            </NavLink>
        </div>
```

Build and test the page to see if the counter starts at 10 and increments by 5.



Passing Data Via Route Paramaters:

Add a razor component called CounterRP.razor to the Pages folder with the following code in it.

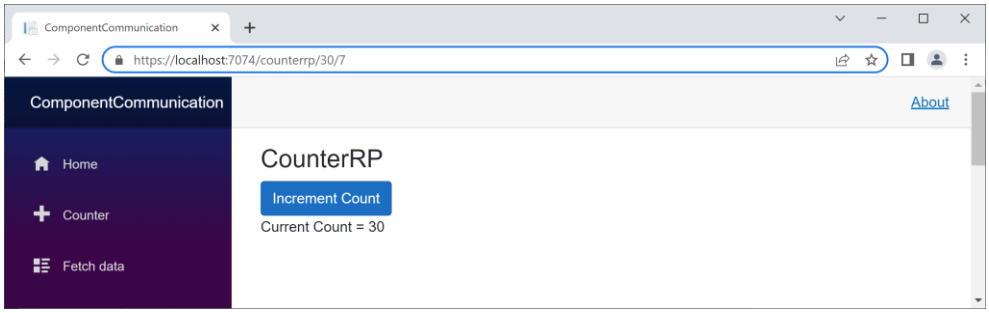
```
@page "/counterrp/{initialcount}"
@page "/counterrp/{initialcount}/{incby}"
<h3>CounterRP</h3>

<button class="btn btn-primary" @onclick="IncrementCount">Increment Count</button>
<br/>
<p>Current Count = @currentCount</p>

@code {
    int currentCount = 0;
    // route parameters need to marked with regular [Parameter] attribute
    // [Parameter] attribute requires the variable to be declared as a property
    [Parameter]
    public string incby { get; set; }
    [Parameter]
    public string initialcount { get; set; }

    protected override void OnInitialized()
    {
        currentCount = int.Parse(initialcount);
    }

    void IncrementCount()
    {
        if (incby != null)
            currentCount = currentCount + int.Parse(incby);
        else
            currentCount = currentCount + 1;
    }
}
```

Note that the incby is made optional by assigning two urls to the page.

```
@page "/counterrp/{initialcount}"
@page "/counterrp/{initialcount}/{incby}"
```

The route parameters of initialcount and incby have to declared as Parameters in the code section.

Test the page by the following url (your port number may be different)
https://localhost:7074/counterrp/30/7
This will cause the counter to start at 30 and increment by 7.



Passing Data via Query String:

Add a razor component to the Pages folder called CounterQS.razor with the following code in it.

```
@page "/counterqs"
@using Microsoft.AspNetCore.WebUtilities
@using Microsoft.Extensions.Primitives
@inject NavigationManager navManager  // for query string or url routing
<h3>Counter Query String</h3>
<br/>
<button class="btn btn-primary" @onclick="IncrementCount">Increment Count</button>
<p>Current Count = @currentCount</p>

@code {
    int currentCount = 0;
    int incby;  // query string parameter
    int initialcount;  // query string parameter

    protected override void OnInitialized()
    {
        StringValues initCount;
        StringValues incBy;
        var uri = navManager.ToAbsoluteUri(navManager.Uri);
        if (QueryHelpers.ParseQuery(uri.Query).TryGetValue("initialcount", out
initCount))
        {
            currentCount = int.Parse(initCount);
        }
        if (QueryHelpers.ParseQuery(uri.Query).TryGetValue("incby", out incBy))
        {
            incby = int.Parse(incBy);
        }
    }

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

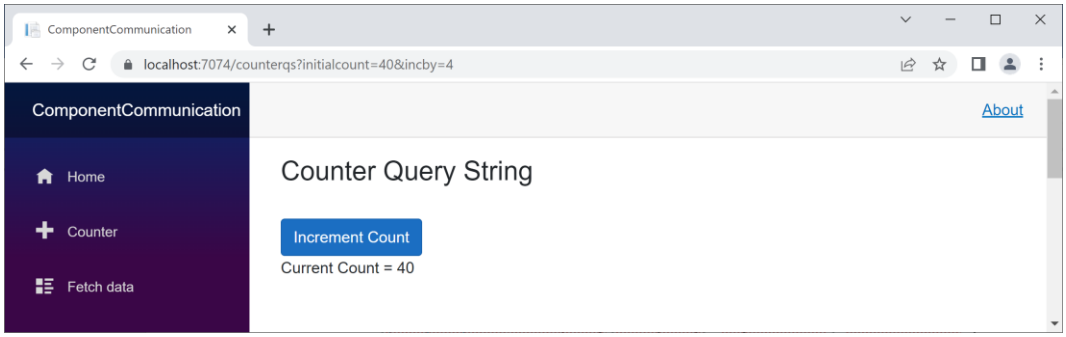
Note that any url related operation is accomplished by injecting NavigationManager in the page.

Notice also how the code in the OnInitialized reads the entire url via the Navigation Manager object, then obtains the query string part, and finally parses the querystring key to obtain its value:

```
        var uri = navManager.ToAbsoluteUri(navManager.Uri);
        if (QueryHelpers.ParseQuery(uri.Query).TryGetValue("initialcount", out
initCount))
```

Test this page by issuing the following url in the browser.

https://localhost:7074/counterqs?initialcount=40&incby=4



The three steps of reading the full url, extracting the query string from it, and then extracting the value of a key from the query string can be made simpler by encapsulating these operations into a single method and extending the NavigationManager class. To accomplish this, add a folder called Utils. Then add a class to it called NavivationManagerExtension with the following code in it.

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.WebUtilities;

namespace ComponentCommunication.Utils
{
    public static class NavigationManagerExtension
    {
        public static bool TryGetQueryString<T>(this NavigationManager navManager,
            string key, out T value)
        {
            var uri = navManager.ToAbsoluteUri(navManager.Uri);
            if (QueryHelpers.ParseQuery(uri.Query).TryGetValue(key, out var
valueFromQueryString))
            {
                if (typeof(T) == typeof(int) && int.TryParse(valueFromQueryString,
out var valueAsInt))
                {
                    value = (T)(object)valueAsInt;
                    return true;
                }
                if (typeof(T) == typeof(string))
                {
                    value = (T)(object)valueFromQueryString.ToString();
                    return true;
                }
                if (typeof(T) == typeof(decimal) &&
decimal.TryParse(valueFromQueryString, out var valueAsDecimal))
                {
                    value = (T)(object)valueAsDecimal;
                    return true;
                }
            }
            value = default;
            return false;
        }
    }
}
```

Add a razor component called CounterQS2.razor to the Pages folder with the following code in it.

```
@page "/counterqs2"
@using ComponentCommunication.Utils
@using Microsoft.AspNetCore.WebUtilities
@using Microsoft.Extensions.Primitives
@inject NavigationManager navManager  // for query string or url routing
<h3>Counter Query String</h3>
<br/>
<button class="btn btn-primary" @onclick="IncrementCount">Increment Count</button>
<p>Current Count = @currentCount</p>

        @code {
    int currentCount = 0;
    int incby;  // query string parameter
    int initialcount;  // query string parameter

    protected override void OnInitialized()
    {
        navManager.TryGetQueryString<int>("initialcount", out currentCount);
        navManager.TryGetQueryString<int>("incby", out incby);
    }

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

This page is similar to CounterQS but uses far less code to extract the values of query string keys of initialcount and incby.

Test this page by issuing the following url.

https://localhost:7074/counterqs2?initialcount=50&incby=5

To demonstrate how a page can create the query string and trigger another page to pass the query string to it, add a razor component called PassDataViaQS.razor with the following code in it. It has a simple form with two text boxes where a user can enter the initial count and the incby values. Once the "Call CounterQS" button is clicked it triggers the CounterQS page and passes the initialcount and the incby by appending it to the url of the target page as a query string.

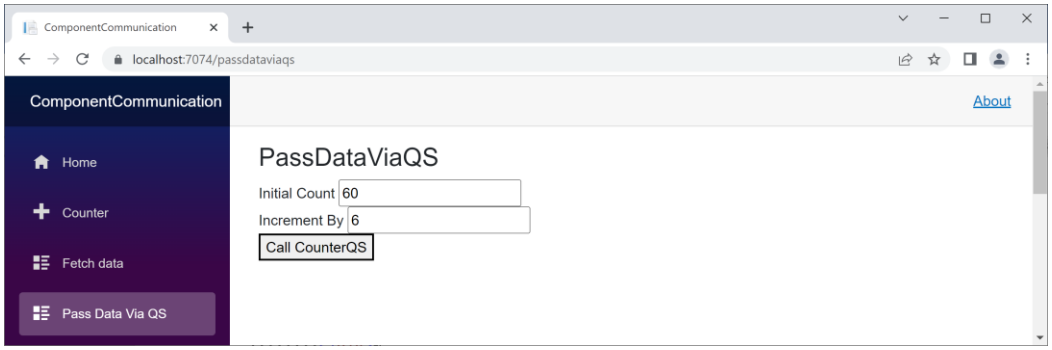
```
@page "/passdataviaqs"
@using Microsoft.AspNetCore.WebUtilities
@inject NavigationManager navManager
<h3>PassDataViaQS</h3>

<EditForm Model="this">
    Initial Count <input @bind="initialcount"/> <br/>
    Increment By <input @bind="incby"/> <br/>
    <button @onclick="CallCounterQS">Call CounterQS</button>
</EditForm>
@code {
        int initialcount = 0;
```

```
        int incby = 0;

        void CallCounterQS()
        {
            var query = new Dictionary<string, string>
            {
                {"initialcount", initialcount.ToString()},
                {"incby", incby.ToString()}
            };
            navManager.NavigateTo(QueryHelpers.AddQueryString("/counterqs2",
query));
        }
}
```

Add a link to this page in the NavMenu.razor as:

```
       <div class="nav-item px-3">
            <NavLink class="nav-link" href="passdataviaqs">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Pass Data
Via QS
            </NavLink>
        </div>
```

Build and test this page.



In Blazor if a hyperlink points to the page itself and passes a querystring value so that the page can be initialized differently, we need to handle the location changed event. Add a component called CounterLC.razor to the Pages folder with the following code in it.

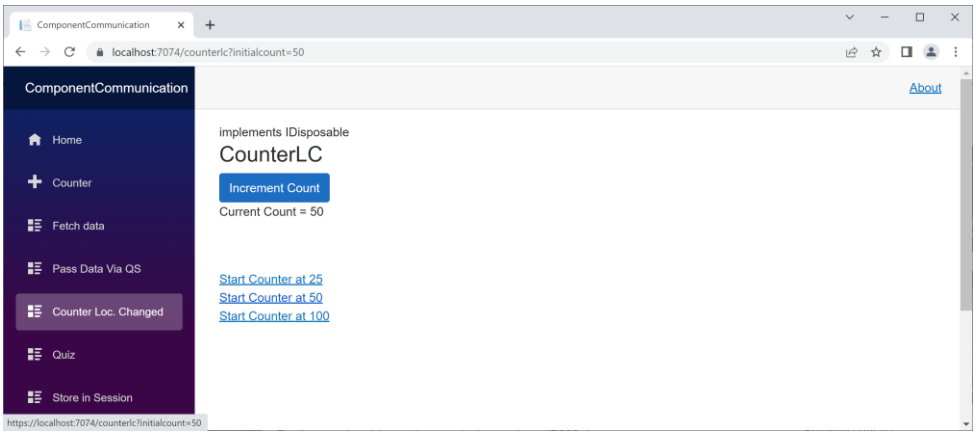
```
@page "/counterlc"
@using ComponentCommunication.Utils
@inject NavigationManager navManager
implements IDisposable

<h3>CounterLC</h3>
<button class="btn btn-primary" @onclick="IncrementCount">Increment Count</button>
<br/>
<p>Current Count = @currentCount</p>
<br/>
<br/>
<a href="/counterlc?initialcount=25">Start Counter at 25</a> <br/>
<a href="/counterlc?initialcount=50">Start Counter at 50</a> <br/>
```

```
<a href="/counterlc?initialcount=100">Start Counter at 100</a> <br/>
@code {
    int currentCount = 0;

    protected override void OnInitialized()
    {
        GetQueryStringValues();
        navManager.LocationChanged += HandleLocationChanged;  // attach event
handler
    }

    void GetQueryStringValues()
    {
        navManager.TryGetQueryString<int>("initialcount", out currentCount);
    }

    void HandleLocationChanged(object sender, LocationChangedEventArgs e)
    {
        GetQueryStringValues();
        StateHasChanged();
    }

    void IncrementCount()
    {
        currentCount = currentCount + 1;
    }

    public void Dispose()
    {
        navManager.LocationChanged -= HandleLocationChanged;
    }
}
```

Add a link to the NavMenu.razor for this page as:

```
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="counterlc">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Counter
Loc. Changed
            </NavLink>
        </div>
```

Then test this page by clicking on the different hyperlinks that pass a query string to the same page.



Parent to Child and Child to Parent Callbacks:

Blazor uses EventCalback type of parameters in a component to trigger events in the parent. The child component declares these parameters and the parent subscribes to the evens by writing the event handler for these callback parameters. To demonstrate an example of it, we will create a MyClock component that will fire 3 events of OnTenSecondsElapsed, OnTwentySecondsEapsed, and TimerStarted. Add a razor component called MyClock.razor to the Pages folder with the following code in it.

```
@page "/myclock"
@using System.Timers
@implements IDisposable

<h3>MyClock</h3>
<div style="border:1px solid red;width:300px;font-size:16pt;font-
weight:bold;color:green">
    <p>@CurrentTime</p>
    <p>Elapsed Seconds: @ElapsedSeconds</p>
    <button @onclick="StartTimer" class="btn btn-success">Start Timer</button>
        
    <button @onclick="StopTimer" class="btn btn-danger">Stop Timer</button>
</div>

@code {
    string? CurrentTime;
    int ElapsedSeconds = 0;
    Timer timer;
    bool startTimer = false;
    [Parameter]
    public EventCallback<int> OnTenSecondsElapsed { get; set; }  // parent will
subscribe to this event
    [Parameter]
    public EventCallback<int> OnTwentySecondsElapsed { get; set; }  // parent will
subscribe to this event
    [Parameter]
    public EventCallback<int> OnTimerStarted { get; set; }  // parent will subscribe
to this event
                                                            // this component will
trigger the above events, and the parent will handle the events

    protected override void OnAfterRender(bool firstRender)
    {  // this gets triggered in the beginning as component is created
        if (firstRender)
        {
            timer = new Timer();
            timer.Interval = 1000;  // milliseconds
            timer.Elapsed += OnTimeInterval;  // event handler for timer
            timer.AutoReset = true;
            timer.Enabled = true;
        }
    }

    async void OnTimeInterval(object sender, ElapsedEventArgs e)
    {
        CurrentTime = DateTime.Now.ToShortTimeString();
        if (startTimer)
```

```
        {
            ElapsedSeconds = ElapsedSeconds + 1;
            if (ElapsedSeconds == 10)
                await InvokeAsync(() =>
OnTenSecondsElapsed.InvokeAsync(ElapsedSeconds)); // fire event
            if (ElapsedSeconds == 20)
                await InvokeAsync(() =>
OnTwentySecondsElapsed.InvokeAsync(ElapsedSeconds)); // fire event
        }
        //StateHasChanged();
        await InvokeAsync(() => StateHasChanged());  // update UI
    }

    async void StartTimer()
    {
        ElapsedSeconds = 0;
        startTimer = true;
        await InvokeAsync(() => OnTimerStarted.InvokeAsync());  // notify parent
timer has started
    }

    public void StopTimer()
    {
        ElapsedSeconds = 0;
        startTimer = false;
    }

    public void Dispose()
    {
        timer?.Dispose();
    }
}
```

Add a razor component to the Pages folder called Quiz1.razor that will use the MyClock as a child component and subscribe to the three events coming from MyClock. Type the following code in Quiz1.razor.

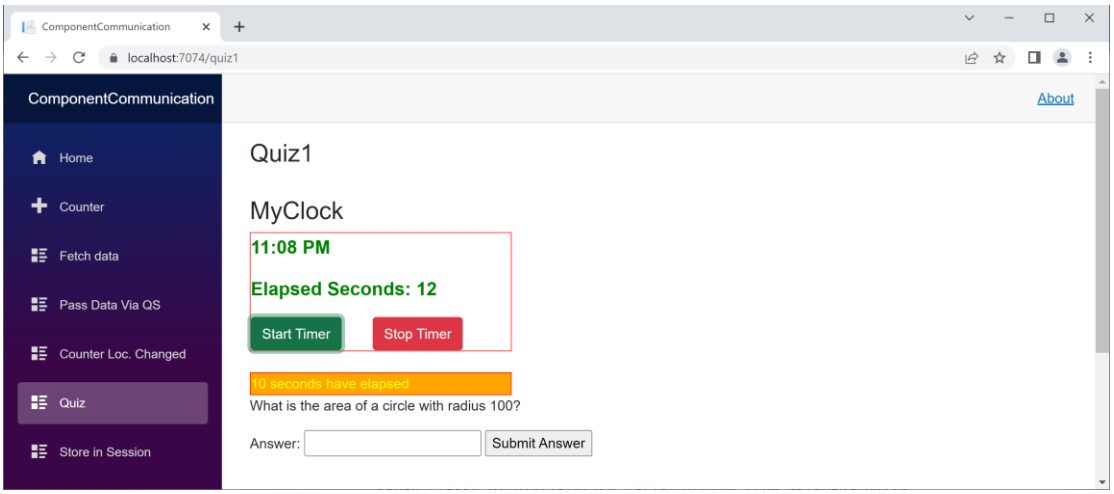
```
@page "/quiz1"
<h3>Quiz1</h3>
<br/>
<MyClock @ref="myclock" OnTenSecondsElapsed="TenSecondsOver"
    OnTwentySecondsElapsed="TwentySecondsOver" OnTimerStarted="TimerStarted"/>
<br/>
<div style="border:1px solid red;background-color:@bcolor;width:300px">
    <span style="color:@textColor">@msg</span>
</div>
<div style="display:@vis">
<p>What is the area of a circle with radius 100?</p>
<form>
    Answer: <input @bind=answer/>
    <input type="button" @onclick="AnswerSubmitted" value="Submit Answer"/>
</form>
</div>
@code {
    MyClock myclock;
    string? answer;
```

```
    string vis = "none";
    string? msg;
    string textColor = "green";
    string? bcolor = "white";

    public void TenSecondsOver(int elapsedSeconds)
    {
        msg = $"{elapsedSeconds} seconds have elapsed";
        textColor = "yellow";
        bcolor = "orange";
    }

    public void TwentySecondsOver(int elapsedSeconds)
    {
        msg = $"{elapsedSeconds} seconds have elapsed";
        textColor = "yellow";
        bcolor = "red";
        myclock.StopTimer();
        vis = "none";
    }

    public void TimerStarted()
    {
        msg = "timer started, you have 20 seconds..";
        textColor = "green";
        bcolor = "white";
        vis = "block";   // display question and the answering text box
    }

    void AnswerSubmitted()
    {
        myclock.StopTimer();
        msg = "Answer submitted..";
        vis = "none";
    }

}
```

Note that as we create the MyClock we add reference to it via the:
`@ref="myclock"`
Attribute so that the parent component Quiz1 can access the methods in MyClock when needed. For example to stop the timer when answer is submitted.
Add a link to Quiz1 in the NavMenu.razor.

```
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="quiz1">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Quiz
            </NavLink>
        </div>
```

Test the Quiz1 page. See the color changes after 10 seconds have elapsed and the quiz question disappearing after 20 seconds. Study the code to see how the quiz question is made visible or invisible by setting the display property of the div to none or block.



Page to Page Communication by Passing Data via the Browser's Memory:

Blazor provides ProtectedLocalStore and ProtectedSessionStore classes to store the temporary data (e.g., shopping cart or user information upon login) in the browser's memory. ProtectedSessionStore is only available as long you stay on the same browser tab, where as ProtectedLocalStore is available even if you close the browser. For storing class objects into the browser's memory, we can serialize a class object into a json string (whch is a key, value pair type of data), then save the json in the browser's memory. Whenever we store data in the browser's memory, we save the json data against a key that we specify. Similary to recover the stored data, we read the data for the same key that was used to store the data, then deserialize the json to a class object. For json serialization and deserialzation, we can use the NewtonSoft.json library from Nuget.

Add a razor component called SetUserInfo.razor to the Pages folder with the following code in it.

```
@page "/setuserinfo"
@using ComponentCommunication.Data
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@using Newtonsoft.Json
@inject ProtectedSessionStorage MyStore
<h3>Set User Info</h3>
<br/>
<button @onclick="StoreUserInfo">Store User Info</button>
<br/>
<p>@Msg</p>

@code {
    string Msg = "";

    void StoreUserInfo()
    {
        UserInfo uinfo = new UserInfo { UserName = "bill", Email = "bill@yahoo.com"
};
        string json = JsonConvert.SerializeObject(uinfo);
        MyStore.SetAsync("UINFO", json);
        Msg = "user info stored in browser session";
    }
```

```
}
```

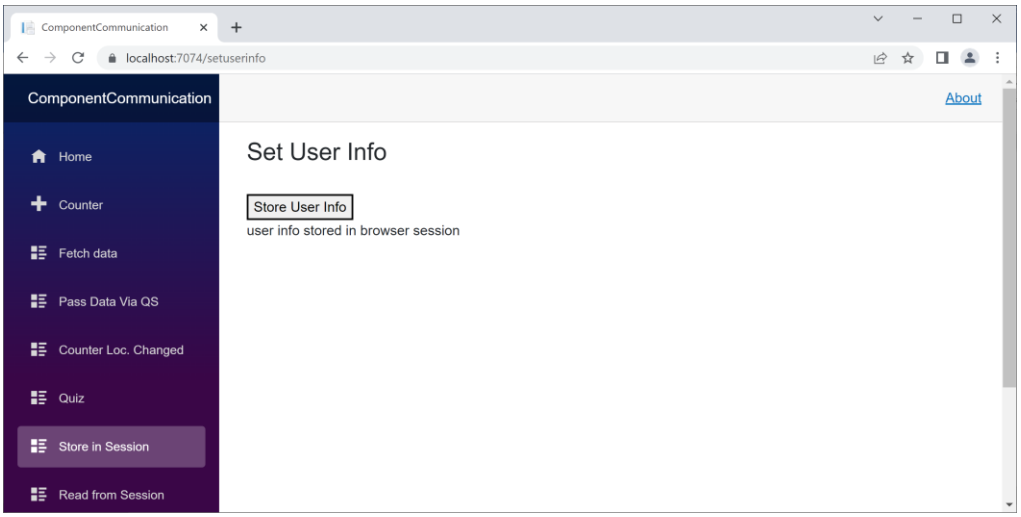
Then add another component that will read the stored data called ReadUserInfo.razor with the following code in it.

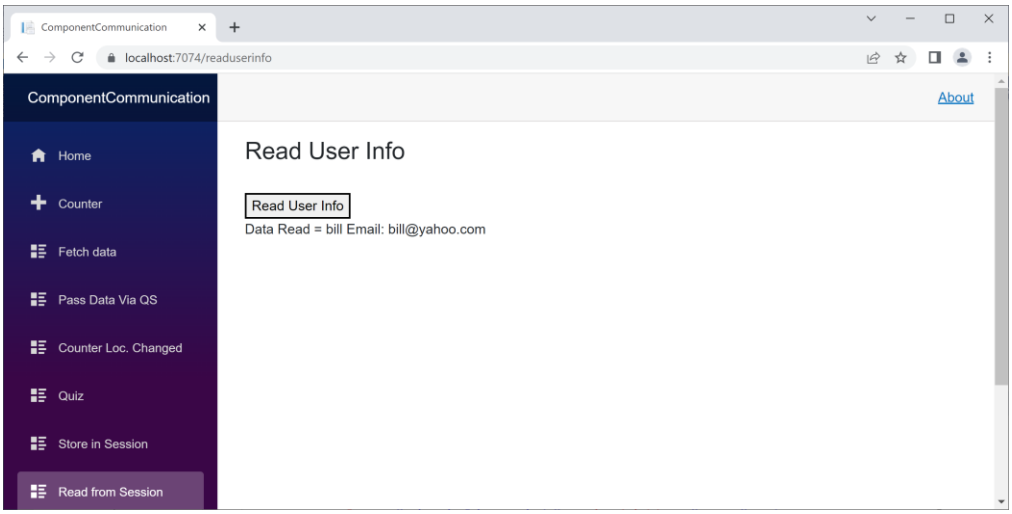
```
@page "/readuserinfo"
@using ComponentCommunication.Data
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@using Newtonsoft.Json
@inject ProtectedSessionStorage MyStore

<h3>Read User Info</h3>
<br/>
<button @onclick="ReadUserInfoFromSession">Read User Info</button>
<br/>
<p>@Msg</p>

@code {
    string Msg = "";
    async void ReadUserInfoFromSession()
    {
        var json = await MyStore.GetAsync<string>("UINFO");
        string jsonstr = json.Value;
        if (jsonstr != null)
        {
            UserInfo uinfo = JsonConvert.DeserializeObject<UserInfo>(jsonstr);
            Msg = "Data Read = " + uinfo.UserName + " Email: " + uinfo.Email;
            await InvokeAsync(() => StateHasChanged());
        }
        else
        {
            Msg = "No data found..";
            await InvokeAsync(() => StateHasChanged());
        }
    }
}
```

Add links to the SetUserInfo and the ReadUserInfo in NavMenu.razor.

```
<div class="nav-item px-3">
            <NavLink class="nav-link" href="setuserinfo">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Store in
Session
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="readuserinfo">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Read from
Session
            </NavLink>
        </div>
```

Test the two pages by first invoking the SetUserInfo and then the ReadUserInfo.





To demonstrate the ProtectedLocalStore which is a longer term memory in the browser, add a page called StoreLocal.razor with the following code in it.

```
@page "/storelocal"
@using ComponentCommunication.Data
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@using Newtonsoft.Json
@inject ProtectedLocalStorage MyStore
<h3>Set User Info</h3>
<br/>
<button @onclick="StoreUserInfo">Store User Info</button>
<br/>
<p>@Msg</p>

@code {
    string Msg = "";

    void StoreUserInfo()
    {
        UserInfo uinfo = new UserInfo { UserName = "sally", Email =
"sally@yahoo.com" };
```

```
        string json = JsonConvert.SerializeObject(uinfo);
        MyStore.SetAsync("UINFOLOCAL", json);
        Msg = "user info stored in local storage";
    }
}
```

To be able to read the Local store, add a razor component to the Pages folder called ReadLocal.razor with the following code in it.

```
@page "/readlocal"
@using ComponentCommunication.Data
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@using Newtonsoft.Json
@inject ProtectedLocalStorage MyStore

<h3>Read User Info</h3>
<br/>
<button @onclick="ReadUserInfoFromLocal">Read User Info</button>
<br/>
<p>@Msg</p>

@code {
    string Msg = "";
    async void ReadUserInfoFromLocal()
    {
        var json = await MyStore.GetAsync<string>("UINFOLOCAL");
        string jsonstr = json.Value;
        if (jsonstr != null)
        {
            UserInfo uinfo = JsonConvert.DeserializeObject<UserInfo>(jsonstr);
            Msg = "Data Read = " + uinfo.UserName + " Email: " + uinfo.Email;
            await InvokeAsync(() => StateHasChanged());
        }
        else
        {
            Msg = "No data found..";
            await InvokeAsync(() => StateHasChanged());
        }
    }
}
```

Add links to the StoreLocal and ReadLocal pages in the NavMenu.razor:

```
<div class="nav-item px-3">
            <NavLink class="nav-link" href="storelocal">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Store Local
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="readlocal">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Read Local
            </NavLink>
        </div>
```

Test the StoreLocal and ReadLocal pages. After storing the data using StoreLocal, copy the url, and close the browser. Then trigger the ReadLocal after launching the browser. You will see that it is able to read the data.