Blazor

ASP520

Module 5

Componentization & Javascript Integration

1 UI Componentization

1.1 Blazor Components

A Blazor application development activities are mostly centered around building and using components. These components are then arranged into parent components and pages. Components can be build to be reusable across multiple locations but even the use of monolithic components can help simplify the development and management of a complex UI. Let us begin by componentizing the user profile page.

Create a **UserDetails** component in the **Shared** folder and move the **MudCard** from the **Profile** page. Any model and methods required can be declared as parameters by attaching the **Parameter** attribute.

Use details component: Shared\UserDetails.razor

```
<MudCard>
    <MudCardHeader>
        <CardHeaderAvatar>
            <MudAvatar Color="Color.Secondary">@Item.Id.Substring(0,1)</MudAvatar>
        </CardHeaderAvatar>
        <CardHeaderContent>
            <MudText Typo="Typo.body1">@Item.Id</MudText>
            <MudText Typo="Typo.body2">@Item.Name</MudText>
        </CardHeaderContent>
    </MudCardHeader>
    <MudCardContent>
        <MudTextField Label="COMPANY" ReadOnly=true Value=@Item.Company />
        <MudTextField Label="POSITION" ReadOnly=true Value=@Item.Position />
        <MudTextField Label="EMAIL" ReadOnly=true Value=@Item.Email />
    </MudCardContent>
    <MudCardActions>
        <MudButton Color=@Color.Primary Variant=@Variant.Filled
            OnClick=@OnLogout>Logout</MudButton>
    </MudCardActions>
</MudCard>

@code {
    [Parameter]public Participant Item { get; set; }
    [Parameter]public EventCallback<MouseEventArgs> OnLogout { get; set; }
}
```

You can now return back to **Profile** page and replace the **MudCard** with **UserDetails** component and pass in the models and methods as parameters.

Using UserDetails component

```
@if (Item != null) {
    <UserDetails Item=@Item OnLogout=@LogoutAsync />
}
```

1.2 Implementing File Upload

We will now implement another component for the user to upload their profile image. Create a **ProfileImageUpload** component in the **Shared** folder. This component will have a delegate parameter named **OnSaveImage** to call the parent component to save the actual image. We will add an **ImageFile** property to assign the selected file, **ImageData** to store the loaded image and **ImageText** for a Base64 encoded version of **ImageData** necessary to display the image in the page.

Parameters and properties: Shared\ProfileImageUpload.razor

```
@code {
    [Parameter]
    public Action<byte[]> OnSaveImage { get; set; }
    IBrowserFile ImageFile { get; set; }
    string ImageText { get; set; }
    byte[] ImageData { get; set; }
}
```

We will add a method named **UploadFileAsync** that will be called each time the user selects a file to upload. If you want to be able to upload multiple files, then you need a **List<IBrowserFile>** property instead to keep multiple selected files. The user can select any file but if you are interested only in image files, convert the selected file to a specific type of image file with a maximum width and height by calling the method **RequestImageFileAsync**. Create a **MemoryStream** and call **OpenReadStream** to open the file for reading. Then call **CopyToAsync** to transfer the file content into the memory stream and call the **ToArray** method to return the content as a byte array. If you need to display the image you have to convert it into a special formatted string. The image itself must be encoded in Base64 format.

Method to be called when file is selected

```
async void UploadFileAsync(IBrowserFile file) {
    ImageFile = await file.RequestImageFileAsync("image/png", 640, 640);
    var buffer = new MemoryStream(); await ImageFile.OpenReadStream().CopyToAsync(buffer);
    ImageData = buffer.ToArray(); buffer.Close();
    ImageText = $"data:image/png;base64, {Convert.ToBase64String(ImageData)}";
    StateHasChanged();
}
```

Method to clear selected file and image

```
void ClearImage() {
    ImageText = null;
    ImageData = null;
    ImageFile = null;
}
```

Method to forward image to parent component to be saved

```
void SaveImage() {
    OnSaveImage?.Invoke(ImageData);
    ClearImage();
}
```

We will use a **MudCard** to encapsulate the content of the component. If an image file is selected, we will use a **MudImage** to display the image. If no file was selected we use a **MudSkeleton** to generate an empty area to represent where the image will be shown. We need to use a **MudFileUpload** to facilitate file selection. It itself does not provide a UI to trigger the file selection process, use a **ButtonTemplate** to create the button to do this. The button must be generated as a HTML **label** so the uploader can locate it and use the **for** property to bind to the upload context. You can add other buttons in the template to perform other operations.

UI for image file uploading

```
<MudCard>
    <MudCardContent>
        @if (ImageFile != null) {
            <MudImage Class="ma-auto" Height="640" Fluid Src=@ImageText />
        }
        else {
            <MudSkeleton SkeletonType=@SkeletonType.Rectangle Height="220px"/>
        }
    </MudCardContent>
    <MudCardActions>
        <MudFileUpload T="IBrowserFile" FilesChanged="UploadFileAsync">
            <ButtonTemplate>
                <MudButton HtmlTag="label" Class="ma-2"
                    Variant=@Variant.Filled" Color=@Color.Primary for=@context>
                    SELECT PROFILE IMAGE
                </MudButton>
                @if (ImageFile != null) {
                    <MudButton Variant=@Variant.Filled
                    Color=@Color.Primary Class="ma-2"
                    OnClick=@SaveImage>SAVE IMAGE</MudButton>
                }
            </ButtonTemplate>
        </MudFileUpload>
    </MudCardActions>
</MudCard>
```

In **Profile** page, we use **MudGrid** to display **UserDetails** and **ProfileImageUpload** on the same row if there is enough space otherwise display as separate rows. You can now select an image file but we still need to write a bit of code to facilitate saving the image into a database. Our **ParticipantDataService** class at the moment does not actually provide a method to save an image. Add the method as shown below.

Using multiple components in page: Pages\user\Profile.razor

```
@if (Item != null) {
    <MudGrid>
        <MudItem xs="12" md="6">
            <UserDetails Item=@Item OnLogout=@LogoutAsync />
        </MudItem>
        <MudItem xs="12" md="6">
            <ProfileImageUpload />
        </MudItem>
    </MudGrid>
}
```

Method to update Photo column: Services\ParticipantDataService.cs

```
public async Task UpdatePhotoAsync(string id, byte[] photo) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var participant = await dc.Participants.FindAsync(id);
        if (participant != null) {
            participant.Photo = photo;
            await dc.SaveChangesAsync();
        }
    }
}
```

We will now add a **SaveImageAsync** method to actually save the image from the UI. The method needs to create notifications so make sure **ISnackbar** is injected. Assign the method to **OnSaveImage** component parameter of **ProfileImageUpload**. You should now be able to select and save the profile image.

Enable access to Snackbar service: Pages\user\Profile.razor

```
@inject ISnackbar Snackbar
```

Method to save image and display result

```
async void SaveImageAsync(byte[] image) {
    try {
        await ParticipantDataService.UpdatePhotoAsync(Item.Id, image);
        Item = await ParticipantDataService.GetParticipant(Item.Id);
        Snackbar.Add("Profile image updated.", Severity.Success);
        StateHasChanged();
    }
    catch (Exception) {
        Snackbar.Add("Error saving profile image.", Severity.Error);
    }
}
```

Assign method to component parameter

```
<ProfileImageUpload OnSaveImage=@SaveImageAsync />
```

We also need to update the **UserDetails** component to display the profile image. First we will add a property to convert the **Photo** byte array property to a special Base64 encoded format for display.

Property to encode byte array for display

```
string ImageText {
    get {
        var text = Convert.ToBase64String(Item.Photo);
        return $"data:image/png;base64, {text}";
    }
}
```

We will update the UI to use **MudImage** component to display the profile image in a **MudAvatar** if the current participant does have a photo. Use **Size** property to change the size of the avatar.

Displaying image avatar

```
<MudAvatar Color=@Color.Secondary Size=@Size.Large>
    @if (Item.Photo == null) { <span>@Item.Id.Substring(0,1)</span> }
    else { <MudImage Src=@ImageText /> }
</MudAvatar>
```

You can actually check the file size and content type before accepting a file upload to ensure the *file is not too large* or is the incorrect type of file. We will now return back to **ProfileImageUpload** component and add **Error** property to store error messages and add a **MudAlert** to display it. We will use a **try catch** block in **UploadFileAsync** method construct the error message. We will add checking for file size and type.

Error property to store error message

```
string Error { get; set; } = string.Empty;
```

Displaying error message

```
<MudCardContent>
    @if (ImageFile != null) { <MudImage Class="ma-auto" Height="640" Fluid Src=@ImageText /> }
    else { <MudSkeleton SkeletonType=@SkeletonType.Rectangle Height="220px"/> }
    @if (Error != string.Empty) { <br />MudAlert Severity=@Severity.Error>@Error</MudAlert> }
</MudCardContent>
```

Check file size and content type

```
async void UploadFileAsync(IBrowserFile file) {
    try {
        Error = string.Empty;
        if (file.Size > 1000000 throw new Exception("File size must not exceed 1MB");
        if (file.ContentType != "image/png")
            throw new Exception("Image file must be in PNG format.");
                 :
    catch (Exception ex) { Error = $"Upload failed. {ex.Message}"; }
}
```

1.3 Implementing Menus

A basic **MudMenu** contains one or more **MudMenuItems**. The items and the menu itself can be considered as buttons. Clicking on the menu button will open the menu. You can use **Label** property to create a text button and **Icon** property to create an icon button. A menu item can act like a link by setting the **Href** property or a button by setting the **OnClick** property. If you create an **ActivatorContent** section, you can put any content here to open the menu and use **ChildContent** section to contain the menu items. The following shows how to use a **MudAvatar** to open the menu.

Creating a menu that is opened with an avatar: Shared\UserDetails.razor

```
<MudMenu>
    <ActivatorContent>
        <MudAvatar ...>...</MudAvatar>
    </ActivatorContent>
    <ChildContent>
        <MudMenuItem OnClick=@(()=>OnSaveImage?.Invoke(null))>Clear Image</MudMenuItem>
    </ChildContent>
</MudMenu>
```

Delegate parameter for save image method

```
[Parameter]public Action<byte[]> OnSaveImage { get; set; }
```

A method to saves the image: Pages\user\Profile.razor

```
<UserDetails Item=@Item OnSaveImage=@SaveImageAsync OnLogout=@LogoutAsync />
```

We will also add a menu to **ProfileImageUpload** component and use the image to activate the menu. We set **PositionAtCursor** so that the menu will appear over the location we click on the image.

Clearing local image: Shared\ProfileImageUpload.razor

```
<MudMenu PositionAtCursor=@true>
    <ActivatorContent>
        <MudImage Class="ma-auto" Height="640" Fluid Src=@ImageText />
    </ActivatorContent>
    <ChildContent>
        <MudMenuItem OnClick=@ClearImage>Clear Image</MudMenuItem>
    </ChildContent>
</MudMenu>
```

1.4 Tables

At the moment the **Profile** component retrieves the **Participant** but not the **Events** of that the user will be participating in. We will now add the following methods to the **ParticipantDataService** that will retrieve also the events for the participant.

Retrieve participant details and events for a specific participant

```
public async Task<Participant> GetParticipantWithEventsAsync(string id) {
    using (var dc = await _dcFactory.CreateDbContextAsync())
        return dc.Participants.Include(p => p.Events).SingleOrDefault(p => p.Id == id);
}
```

Retrieve participant and events for current user

```
public async Task<Participant> GetCurrentParticipantWithEventsAsync() {
    var state = await _authenticationStateProvider.GetAuthenticationStateAsync();
    if (state != null) return await GetParticipantWithEventsAsync(state.User.Identity.Name);
    return null;
}
```

We will update **Profile** page to call the above method so that we can also show the **Events** the user is participating in the page. We also need the **EventDataService** to perform event-related operations. Then add a **LeaveEventAsync** method that can be called to remove the event from the current user.

Update Profile page to call the above method: Pages\user\Profile.razor

```
protected override async Task OnInitializedAsync() {
    Item = await ParticipantDataService.GetCurrentParticipantWithEventsAsync();
}
```

```
async void SaveImageAsync(byte[] image) {
    try {
            await ParticipantDataService.UpdatePhotoAsync(Item.Id, image);
            Item = await ParticipantDataService.GetParticipantWithEventsAsync(Item.Id);
                    :
}
```

Get access to EventDataService

```
@page "/user/profile"
@attribute [Authorize]
@inject ParticipantDataService ParticipantDataService
@inject EventDataService EventDataService
@inject NavigationManager Navigation
@inject ISnackbar Snackbar
```

Method to leave event

```
async void LeaveEventAsync(Event eventItem) {
    try {
        await EventDataService.RemoveParticipantAsync(eventItem.Id, Item.Id);
        Item = await ParticipantDataService.GetParticipantWithEventsAsync(Item.Id);
        StateHasChanged();
    }
    catch (Exception) {
        Snackbar.Add($"Error leaving event '{eventItem.Id}'.", Severity.Error);
    }
}
```

Add an **EventList** component into **Shared** folder that uses a **MudTable** to display the list of events assigned to **Items** parameter and each row has a **MudIconButton** that the user can click on to use the **OnLeaveEvent** parameter to remove the user from the event.

Participant's event list component: Shared\EventList.razor

```
<MudTable Items=@Items>
    <HeaderContent>
        <MudTh>
            <MudTableSortLabel
                SortBy=@(new Func<Event,object>(e => e.From))>
                Date
            </MudTableSortLabel>
        </MudTh>
        <MudTh>ID</MudTh>
        <MudTh>Title</MudTh>
        <MudTh></MudTh>
    </HeaderContent>
    <RowTemplate>
        <MudTd>@context.From</MudTd>
        <MudTd>@context.Id</MudTd>
        <MudTd>@context.Title</MudTd>
        <MudTd>
            <MudIconButton
                Icon=@Icons.Material.Rounded.Delete
                OnClick=@(()=>OnLeaveEvent?.Invoke(context)) />
        </MudTd>
    </RowTemplate>
</MudTable>
```

```
@code {
    [Parameter]public IEnumerable<Event> Items { get; set; }
    [Parameter]public Action<Event> OnLeaveEvent { get; set; }
}
```

We will now add the above component into the **Profile** page and assign the required parameters. We use **MudExpansionPanels** to reveal or collapse optional content. It is also commonly called as an accordion in many other UI frameworks.

Adding EventList component to profile page: Pages\user\Profile.razor

```
<MudGrid>
    <MudItem xs="12" md="6">
        <UserDetails Item=@Item
            OnSaveImage=@SaveImageAsync
            OnLogout=@LogoutAsync />
    </MudItem>
    <MudItem xs="12" md="6">
        <ProfileImageUpload OnSaveImage=@SaveImageAsync />
    </MudItem>
    <MudItem xs="12">
        <MudExpansionPanels>
            <MudExpansionPanel>
                <TitleContent><MudText Typo=@Typo.h5>Events</MudText></TitleContent>
                <ChildContent>
                    <EventList Items=@Item.Events OnLeaveEvent=@LeaveEventAsync />
                </ChildContent>
            </MudExpansionPanel>
        </MudExpansionPanels>
    </MudItem>
</MudGrid>
```

Currently clicking on the delete icon automatically removes the participant from the event. You can use a dialog to ask the user to confirm first before removing. However rather than implementing your own dialog, you can use a **MessageBox** dialog that is already provided by the dialog service. We will add the code in **LeaveEventAsync** so it will always appear regardless of which component displays the event list. We need access to **IDialogService** to do this. Call **ShowMessageBox** with the title and the confirmation message. The result will be **true** if the user confirms.

Get access to dialog service: Pages\user\Profile.razor

```
@inject IDialogService DialogService
```

Use MessageBox provided by dialog service

```
async void LeaveEventAsync(Event eventItem) {
    try {
        bool? result = await DialogService.ShowMessageBox(
            "Warning", $"Leave event '{eventItem.Id}'?");
        if (result != true) return;
                    :
```

You can use **MudMessageBox** to customize your own message box dialog. Call the **Show** method to make it appear and wait for the result. There is an example of doing this in the documentation for *MessageBox* at the MudBlazor site.

Rather than using a button or icon to perform operations directly on each item in the table, you can use the **OnRowClick** event on a **MudTable** to detect the user clicking on an item and use a dialog or a separate section to display details and provide more actions. In **EventList**, declare a **Item** property to assign a single **Event** and then add a **RowSelected** method to capture the selected item and update the **Item** property.

Update component to support item select: Shared\EventList.razor

```
Event Item { get; set; }

void RowSelected(TableRowClickEventArgs<Event> e) {
    Item = e.Item;
}
```

Create a **MudGrid** with two **MudItem**. Move the **MudTable** into the first **MudItem** and make sure to set **T** to the type name of each item and then assign **RowSelected** method to **OnRowClick** event. The second **MudItem** will only appear if an item has been selected. It contains a **MudCard** that shows item details and provide a button to trigger the **OnLeaveEvent** delegate.

Updated UI for item selection

```
<MudGrid>
    <MudItem xs="12" md="8">
        <MudTable T="Event" Items=@Items OnRowClick="RowSelected">
                    :
        </MudTable>
    </MudItem>
    @if (Item != null) {
        <MudItem xs="12" md="4">
            <MudCard>
                <MudCardContent>
                    <MudTextField Label="ID" Value=@Item.Id ReadOnly />
                    <MudTextField Label="TITLE" Value=@Item.Title ReadOnly />
                </MudCardContent>
                <MudCardActions>
                    <MudButton Color=@Color.Primary Variant=@Variant.Filled
                        OnClick=@(()=>OnLeaveEvent?.Invoke(Item))>
                        LEAVE EVENT
                    </MudButton>
                </MudCardActions>
            </MudCard>
        </MudItem>
    }
</MudGrid>
```

To highlight the row you click, set **SelectOnRowClick** to **true**. You can then assign a method **RowClassFunc** property to a method that will return the style class for each row. We can return a different style class for the item selected.

Highlight row selection

```
<MudTable T="Event" Items=@Items
    SelectOnRowClick=@true
    RowClassFunc=@GetRowClass
    OnRowClick=@RowSelected>
            :
```

Returning custom style class for selected row

```
string GetRowClass(Event item, int row) {
    if (Item == item) return "selected";
    return string.Empty;
}
```

An example style class for row

```
<style>
    .selected { background-color: #1E88E5 !important; }
    .selected > td { color: white !important; }
    .selected > td .mud-input {  color: white !important; }
</style>
```

Rather than using **OnRowClick**, you can also bind the **SelectedItem** property on the **MudTable** to the **Item** property. Since **SelectOnRowClick** is enabled, clicking on the row will automatically select the item.

Binding table SelectedItem property

```
<MudTable T="Event" Items=@Items
    @bind-SelectedItem=@Item
    RowClassFunc=@GetRowClass>
```

To support multiple selections, add a **HashSet** to store multiple selected items. Then add a method to leave the event for every item in the **HashSet**. Bind the **HashSet** to the **SelectedItems** property on the table and set **MultiSelection** to **true**.

Code to support multiple selections

```
HashSet<Event> SelectedItems { get; set; } = new();

void LeaveSelectedEvents() {
    if (OnLeaveEvent != null) {
        foreach (var item in SelectedItems)
            OnLeaveEvent.Invoke(item);
    }
}
```

Enable multi-selection on table

```
<MudTable T="Event" Items=@Items
    SelectOnRowClick=@true
    MultiSelection=@true
    @bind-SelectedItems=SelectedItems
    @bind-SelectedItem=Item
    RowClassFunc=@GetRowClass>
```

Add a **ToolBarContent** section into the table to display a button that calls the above **LeaveSelectedEvents** method. We can check the **SelectedItems** HashSet to make sure that there is at least one item selected for the button to appear.

Button to process selected items

```
<ToolBarContent>
    @if(SelectedItems.Count > 0) {
        <MudButton Color=@Color.Primary Variant=@Variant.Filled
            OnClick=@LeaveSelectedEvents>
            LEAVE EVENTS
        </MudButton>
    }
</ToolBarContent>
```

You might not want to display a table if there are no items. Alternatively you can still display the table but add a **NoRecordsContent** that will only be displayed when the table does not have any items to show.

Display alternative content in empty table

```
<NoRecordsContent>
    <MudAlert Class="ma-4" Severity=@Severity.Info>
        You have not participated in any scheduled events.
        Click <MudLink Href="/events">here</MudLink> to
        view the scheduled events.
    </MudAlert>
</NoRecordsContent>
```

Instead of using a **MudTable**, there is a more advanced **MudDataGrid** component. However this component is experimental and still under development so you should stick with **MudTable** until this component is finalized.

1.5 Tabs

Currently the **Profile** page is showing all components at the same time in a **MudGrid** but we can also use a **MudTabs** and place each component in a **MudTabPanel**. Only one panel will be visible at one time. The user can easily switch between the panels using the links generated by **MudTabs** component.

Using MudTabs and MudTabPanel components: Pages\user\Profile.razor

```
<MudTabs>
    <MudTabPanel Text="USER DETAILS">
        <UserDetails Item=@Item
            OnSaveImage=@SaveImageAsync
            OnLogout=@LogoutAsync />
    </MudTabPanel>
    <MudTabPanel Text="IMAGE PROFILE">
        <ProfileImageUpload OnSaveImage=@SaveImageAsync />
    </MudTabPanel>
    <MudTabPanel Text="EVENTS">
        <EventList Items=@Item.Events OnLeaveEvent=@LeaveEventAsync />
    </MudTabPanel>
</MudTabs>
```

If you want to select a panel from code, declare an **int** field or property and bind it to the **ActivePanelIndex** on **MudTabs** component. Changing it will automatically cause the corresponding **MudTabPanel** to be selected. The index starts at **0**.

Property to bind to ActivePanelIndex

```
int CurrentPanel { get; set; }
```

Binding property in MudTabs component

```
<MudTabs @bind-ActivePanelIndex=CurrentPanel>
                    :
</MudTabs>
```

For example, we may want to switch from **ProfileImageUpload** to **UserDetails** if an image has been uploaded successfully since the image is only shown in **UserDetails**. Change the **CurrentPanel** property in the code will update the UI. There is no need to call **StateHasChanged** unless you have issues with other components.

Switching panels from code

```
async void SaveImageAsync(byte[] image) {
    try {
        await ParticipantDataService.UpdatePhotoAsync(Item.Id, image);
        Item = await ParticipantDataService.GetParticipantWithEventsAsync(Item.Id);
        Snackbar.Add("Profile image updated.", Severity.Success);
        CurrentPanel = 0;
        StateHasChanged();
    }
    catch (Exception) {
        Snackbar.Add("Error saving profile image.", Severity.Error);
    }
}
```

1.5 Maintaining Component State

There are many ways to maintain state. Persistent state can be maintained by storing it in a database. Application state that only needs to be maintained by the application while it is still running can be implemented a singleton service to store state. You can use caching for state that can expire. To store session state on the server you can use a **DistributedMemoryCache**. A cookie is required to maintain the link between the browser session and the server session. Once session storage is configured as shown below, you can access the **Session** object from **HttpContext** to store and retrieve any value or object. Make sure to call **Clear** on the **Session** when the user logs out. If the session cookie expires, a new session will be created.

Implement server session storage

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(options => {
    options.Cookie.Name = "newrise_session";
    options.IdleTimeout = TimeSpan.FromMinutes(20);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
var app = builder.Build();
app.UseSession();
```

The problem with storing session state on the server is it could potentionally use up a lot of memory when there could be thousands of simultaneous sessions. It is also not applicable to Blazor server applications since **HttpContext** is needed for both cookies and access to the session state. Razor components have no access to **HttpContext** as they are activated through SignalR and not by HTTP requests.

Alternatively, we can store state on the browser. To keep a persistent state, use local storage, and for session state you can use session storage. Session state will only be maintained while the browser window is opened. If the information is private, we can use protected session storage as we have used for our custom authentication state provider. You can check this in the Application page of the Developer Tools available on your web browser. The only issue is security as the state is kept on the client side. Thus you must make sure to encrypt private and sensitive state by using either the **ProtectedSessionStorage** or **ProtectedLocalStorage**. Do note that these services are only available for Blazor Server applications since need to use the Data Protection API on the server. You can use third-party packages like **Blazored.LocalStorage** and **Blazored.SessionStorage** for Blazor WebAssembly applications. These stores do not encrypt stored values so you need to manually encrypt the values before storing them if you need to protect the values.

Notice that your search text in the **Events** page disappears when you switch pages. We can maintain state across pages by storing it in session or local storage. Below is shown the steps to take to maintain the search text in session storage.

Get access to storage service: Pages\Events.razor

```
@inject ProtectedSessionStorage SessionStore
```

Declare field to cache the value

```
string _searchText = string.Empty;
```

Use property to store and update the value

```
string SearchText {
    get => _searchText;
    set => SessionStore.SetAsync("Events.SearchText", _searchText = value);
}
```

You can do the same thing for **CurrentPanel** in **Profile** page. Regardless of the panel you selected in the page, everytime you navigate away from the page, the panel that was active is already forgotten. The active panel will always be the first panel when you navigate back to the page.

2 Javascript Interoperation

2.1 Calling JavaScript

As you have seen that we can implementing a complete web application without using any Javascript. Any Javascript required is already implemented by Blazor components and libraries provided by Microsoft and third-party. However if your application needs to interoperate with Javascript code, you can still do so. Use the **IJSRuntime** service to call Javascript functions. You can inject the service into any component.

Inject Javascript runtime service: TestJS1.razor

```
@page "/js1"
@inject IJSRuntime JS
<MudGrid Class="pa-8">
    <MudItem>
        <MudButton Variant="Variant.Filled" Color="Color.Primary"
            OnClick="ShowAlertAsync">Show Alert</MudButton>
        <MudText Class="pt-4">Result:@result</MudText>
    </MudItem>
</MudGrid>
```

The following is a Javascript function that simply displays an alert box. Functions can accept zero or more arguments and optional returns a value. The function below will return a boolean result of true when completed.

Javascript function

```
<script>
    function show_alert(message) { alert(message); return true;  }
</script>
```

Use the **IJSRuntime** service to call Javascript functions. Use **InvokeAsync** to call all functions that has a return value and specify the return value type. If no return value is expected then use **InvokeVoidAsync** method instead. Note all functions are called asynchronously. Pass in the name of the function and any parameters must be passed as an object array.

Calling Javascript functions

```
@code {
    bool result;
    async Task ShowAlertAsync() {
        result = await JS.InvokeAsync<bool>(
            "show_alert", new object[] { "Hello Blazor!" });
    //  StateHasChanged();
    }
}
```

2.2 Calling Blazor

To callback C# methods from Javascript, you can a use a **DotNet** object provided by Blazo to call static methods. Use **invokeMethod** or **invokeMethodAsync** to callback methods. Pass in the name of the assembly, static method name and any additional arguments. The type of arguments must be JSON serializable.

To call object methods, you need to create a **DotNetObjectReference** wrapper over the object and pass this object over to Javascript. To do multiple callbacks, this object can be assigned to a Javascript variable. Even though this object can be pre-created in **Initialize** or **InitializeAsync**, you cannot call Javascript functions until the page is rendered in **AfterRender** or **AfterRenderAsync**. We only need to pass in the object once, so we will do it only on **firstRender**. You can then use this object in Javascript to callback instead of **DotNet** object.

Injecting Javascript runtime services: TestJS2.razor

```
@page "/js2"
@inject IJSRuntime JS

<MudGrid Class="pa-8">
    <MudItem>
        <button id="btn1">Set Message</button>
        <MudText>Message:@message</MudText>
    </MudItem>
</MudGrid>
```

The following are Javascript functions that will interoperate with C# methods. We call **set_instance** function from C# to pass the DotNet object to access from Javascript. We also attach a client-slide event handler function **set_message** to the HTML button in the above page. This function will call back C# method **SetMessage** when button is clicked.

Javascript interop functions

```
<script>
    var instance = null;

    function set_message() {
    //  Dotnet.invokeMethodAsync("Newrise","SetMessage", "Hello Javascript!");
        instance.invokeMethodAsync("SetMessage", "Hello Javascript!");
        alert("message sent!");
    }

    function set_instance(obj) {
        instance = obj;
        const button1 = document.getElementById("btn1");
        button1.addEventListener("click", set_message);
        alert("instance set!");
    }
</script>
```

Any C# method that you want to allow to be callable from Javascript must have the **JSInvokable** attribute attached.

In the following C# code, we will create a **DotNetObjectReference** wrapper over the current object in **OnAfterRender** method. This object is created once and passed to the **set_instance** Javascript function on first rendering of the page. We will add the **SetMessage** method to be called from Javascript so it needs **JSInvokable** attribute attached. Blazor does not know when this function is called so you probably need to use **StateHasChanged** to force the UI to update properly.

C# interop methods

```
@code {

    private DotNetObjectReference<TestJS2> this_obj;

    protected override void OnAfterRender(bool firstRender) {
        if (firstRender) {
            this_obj = DotNetObjectReference.Create(this);
            JS.InvokeVoidAsync("set_instance", new object[] { this_obj });
        }
    }

    string message;

    [JSInvokable]
    public void SetMessage(string text) {
        message = text; StateHasChanged();
    }
}
```

The **DotNetObjectReference** wrapper needs to be disposed. Blazor components can be disposable by implementing **IDisposable** or **IDisposableAsync** interfaces. Then add either **Dispose** method or **DisposeAsync** method depending on which interface and use it to dispose of the DotNet object if it has been created.

Implementing a disposable Blazor component

```
@implements IDisposable
```

Implementing disposable

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