ASP530-A

Module 4

Using MudBlazor Services & Components

1 Input Validation

1.1 Data Annotations

In an **EditForm**, you can use a **DataAnnotationsValidator** to automatically validate input fields based on validation attributes attached to fields or properties. You can use **ValidationMessage** to display validation error message per field or property by using **For** property to an expression that identifies the field or property. You can also use a **ValidationSummary** to display all error messages.

ASP.NET validation components

```
<EditForm>
    <DataAnnotationsValidator />
            :
    <div>
        <label for="userid">User ID</label>
        <InputText id="userid" @bind-Value=Item.Id />
        <ValidationMessage For=@(()=>Item.Id) />
    </div>
            :
    <div>
        <ValidationSummary />
    <div>
            :
</EditForm>
```

In MudBlazor you can add **ValidationMessage** to input components by applying the **For** property directly on a MudBlazor input component.

Adding ValidationMessage to MudBlazor components

```
<MudTextField Label="USER ID *" @ref=IdField @bind-Value=Item.Id
    For=@(()=>Item.Id) MaxLength="40" AutoFocus />
```

In registration form we have a **ConfirmPassword** property that we wish to validate. However attaching a validation attribute directly to this element does not work since it is not a member of the **Model** assigned to **EditForm**. This is where a view model will be necessary. A view model is a model use for data presentation or input and not for data storage.

Attaching validation attribute to data element

```
[Required]
string ConfirmPassword { get; set; }
```

Since we already have a **Participant** data model, we can extend it into a view model rather than creating a new view model from scratch. A benefit of doing so is the view model remains compatible with **Participant**.

Implementing a view model: ViewModels\NewParticipant.cs

```
using Newrise.Models;
using System.ComponentModel.DataAnnotations;

namespace Newrise.ViewModels {
    public class NewParticipant : Participant {
        [Required]
        public string ConfirmPassword { get; set; }
    }
}
```

We can now update **Register.razor** and replace **Participant** with the view model. It is now possible to remove the **ConfirmPassword** property and then replace its use with **Item.ConfirmPassword** instead. Add the **For** property in the input component to attach the validation error message.

Replace data model with view model

```
NewParticipant Item { get; set; } = new();
```

Binding to property in the view model for validation

```
<MudTextField Label="CONFIRM PASSWORD *" @bind-Value=Item.ConfirmPassword
    InputType=@InputType.Password MaxLength="8"
    For=@(()=>Item.ConfirmPassword) />
```

Update code to use view model member

```
async Task HandleSubmitAsync(EditContext context) {
    try {
        IsSubmitting = true;
        Error = string.Empty;
        if (Item.Password != Item.ConfirmPassword)
            throw new Exception("Passwords do not match.");
                :
```

Notice the default error message will automatically use the property name. This may not be desired so you can attach a custom error message instead. Alternatively if you still want the default message, use a **Display** attribute and use the **Name** property to change the member's name only for display purposes.

Customizing default error message

```
[Required(ErrorMessage = "Confirmation password is required.")]
public string ConfirmPassword { get; set; }
```

Customize display name

```
[Required, Display(Name = "Confirmation Password")]
public string ConfirmPassword { get; set; }
```

Currently we are still comparing passwords by using code, but you can replace this by using **Compare** validation attribute. Pass the name of the element to compare to. If you do not want to hard-code the name, use the **nameof** operator. You can customize the error message just like all the other validation attributes.

Using Compare validation attribute

```
[Required, Display(Name = "Confirmation Password")]
[Compare(nameof(Password), ErrorMessage = "Passwords do not match")]
public string ConfirmPassword { get; set; }
```

1.2 Custom Validation Attribute

You can use **RegularExpression** attribute to validate a password by combining parts shown below to form the validation part. Then add the range of acceptable characters followed by mininum, maximum number of characters.

Password regular expression validation parts

```
(?=.*[a-z])         Must have at least one lowercase character
(?=.*[A-Z])         Must have at least one uppercase character
(?=.*\d)            Must have at least one digit
(?=.*[$@$!%*?&])    Must have one of the specified symbols
```

Combined expression

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[$@$!%*?&])[A-Za-z\d$@$!%*#?&]{8,}$
```

Using RegularExpression validation attribute

```
[RegularExpression(
    @"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[$@$!%*?&])[A-Za-z\d$@$!%*#?&]{8,}$",
    ErrorMessage = "{0} is not correctly formatted.")]
public string Password { get; set; }
```

Since we can implement a custom validation attribute, let us create a reusable one for not only our current application but can be used for other applications as well. Add a **Validators** folder. In the folder, first add in a **ValidatePasswordOptions enum** for configuring password validate option flags. You must use bit values to construct a flag based enum.

Enum for validate password bit flags: Validators\ValidatePasswordOptions.cs

```
namespace Newrise.Validators {
    public enum ValidatePasswordOptions {
        None = 0,
        HasUpperCase = 1,
        HasLowerCase = 2,
        HasDigit = 4,
        HasSymbol = 8,
        All =
            HasUpperCase |
            HasLowerCase |
            HasDigit |
            HasSymbol
    }
}
```

Then add a **PasswordAttribute** class and extend it from **ValidationAttribute** and override **IsValid** and **FormatErrorMessage** methods. The following code shows how we can implement custom validation.

Methods to override

```
public override bool IsValid(object value) {
    return base.IsValid(value);
}

public override string FormatErrorMessage(string name) {
    return base.FormatErrorMessage(name);
}
```

We will first define a list of fields. The **_options** field is to store the configured value where the **Password** attribute is attached while **_result** stores the validation result. To be valid the **_result** must match the **_options**. The **_allowedSymbols** field will contain all the accepted symbols if **_options** contain **HasSymbol** flag. By default, the password cannot contain spaces, but it can be switched off using **_allowSpace**. The **_minLength** field can be used for customizing the minimum length of password. We can add a **_maxLength** as well but this can be controlled on the input component.

Fields to override

```
public readonly ValidatePasswordOptions _options;
public ValidatePasswordOptions _result;
public string _allowedSymbols = "~`!@#$%^&*-_+=|\\:;\"',.?/(){}[]<>";
public bool _allowSpace = false;
public int _minLength = 8;
public string _error = string.Empty;
```

To allow user to customize validation, we will use properties to expose those fields we allow the user to customize. Customization can be done through constructors as well. We can also expose read-only properties that can be accessed but not changed using a **get** method without a **set** method.

Allow customization of fields

```
public string AllowedSymbols {
    get => _allowedSymbols;
    set => _allowedSymbols = value;
}

public bool AllowSpace {
    get => _allowSpace;
    set => _allowSpace = value;
}

public int MinLength {
    get => _minLength;
    set => _minLength = value;
}
```

Allow access to options, result and error message

```
public ValidatePasswordOptions Options { get => _options; }
public ValidatePasswordOptions Result { get => _result; }
public string Error { get => _error; }
```

We will provide two constructors; the default constructor will enable all the validation options while the second constructor allow customized options. We will then write the code for **IsValid** and **FormatErrorMessage**.

Adding constructors

```
public PasswordAttribute() {
    _options = ValidatePasswordOptions.All;
}

public PasswordAttribute(ValidatePasswordOptions options) {
    _options = options;
}
```

Implementing validation

```
public override bool IsValid(object value) {
    var password = value.ToString();
    _result = ValidatePasswordOptions.None;
    if (password.Length < _minLength) {
        _error = "{0} must have at least {1} characters.";
        return false;
    }
    if (!_allowSpace && password.Contains(' ')) {
        _error = "{0} cannot contain spaces.";
        return false;
    }
    foreach (var code in password) {
        if (Char.IsAsciiLetterLower(code))
            _result |= ValidatePasswordOptions.HasLowerCase; else
        if (Char.IsAsciiLetterUpper(code))
            _result |= ValidatePasswordOptions.HasUpperCase; else
        if (Char.IsAsciiDigit(code))
            _result |= ValidatePasswordOptions.HasDigit; else
        if (_allowedSymbols.Contains(code)) _result |= ValidatePasswordOptions.HasSymbol;
    }
    if (_options.HasFlag(ValidatePasswordOptions.HasLowerCase) &&
        !_result.HasFlag(ValidatePasswordOptions.HasLowerCase)) {
        _error = "{0} must have at least one lowercase character.";
        return false;
    }
    if (_options.HasFlag(ValidatePasswordOptions.HasUpperCase) &&
        !_result.HasFlag(ValidatePasswordOptions.HasUpperCase)) {
        _error = "{0} must have at least one uppercase character.";
        return false;
    }
    if (_options.HasFlag(ValidatePasswordOptions.HasDigit) &&
        !_result.HasFlag(ValidatePasswordOptions.HasDigit)) {
        _error = "{0} must have at least one digit.";
        return false;
    }
    if (_options.HasFlag(ValidatePasswordOptions.HasSymbol) &&
        !_result.HasFlag(ValidatePasswordOptions.HasSymbol)) {
        _error = "{0} must have at least one symbol.";
        return false;
    }
    _error = string.Empty;
    return true;
}
```

Formatting the error message

```
public override string FormatErrorMessage(string name) {
    return string.Format(_error, name, _minLength);
    }
}
```

You can now attach **PasswordAttribute** to any password element. The default values should work well. If you want to customize it, pass in the **ValidatePasswordOptions** and optionally set other properties as shown below.

Attaching password attribute to model: Models\Participant.cs

```
[Password]
public string Password { get; set; }
```

Default customizable values

```
[Password(
    ValidatePasswordOptions.All,
    AllowedSymbols = "~`!@#$%^&*-_+=|\\:;\"',.?/(){}[]<>",
    AllowSpace = false,
    MinLength = 8
)]
```

Of course, you should provide UI to inform the user what are the password rules. This can be implemented as a separate component so you can use in anywhere. It will also be very easy for you to make changes or improvements to the component and rest of the application will be updated with the changes automatically.

Creating a password rules component: Shared\PasswordRules.razor

```
<MudAlert Severity=@Severity.Warning>
    For a valid password, it must have all the following:
    <ul>
        <li>At least one uppercase character</li>
        <li>At least one lowercase character</li>
        <li>At least one digit</li>
        <li>At least one symbol</li>
        <li>Minimum of 8 characters</li>
    </ul>
</MudAlert>
```

Using component in registration: Pages\user\Register.razor

```
<PasswordRules />
<MudStack Row="true">
    <MudTextField Label="PASSWORD *" @bind-Value=Item.Password
        InputType=@InputType.Password For=@(()=>Item.Password) MaxLength="8" />
    <MudTextField Label="CONFIRM PASSWORD *" @bind-Value=Item.ConfirmPassword
        InputType=@InputType.Password MaxLength="8"
        For=@(()=>Item.ConfirmPassword) />
</MudStack><br />
```

1.3 MudBlazor Form

If you are not using data annotations for input validation, you can use the **MudForm** component as in our current **Login** Razor page. Add a state field or property to store validation state and bind it to the **IsValid** property of the form. You can then set the login button's **Disabled** property to validation state to disable it if the validation state is **false**.

Validation state property: Pages/user/Login.razor

```
bool IsValid { get; set; }
```

Binding form validation state

```
<MudForm @bind-IsValid=IsValid>
        :
</MudForm>
```

Disable button on validation failure

```
<MudButton Variant="Variant.Filled"
    Color="Color.Primary" Class="ml-auto"
    Disabled=@(!IsValid) OnClick="LoginAsync">LOGIN</MudButton>
```

For all required fields, just set **Required** property of the input component to **true** and use **RequiredError** to assign a custom error message. The validation error messages should appear without setting the **For** property.

Enabling required field validation

```
<MudTextField Label="USER ID/EMAIL"
    InputType="InputType.Email" @bind-Value=UserName
    Required=@true RequiredError="User ID or Email required." /><br />

<MudTextField Label="PASSWORD"
    InputType="InputType.Password" @bind-Value=Password
    Required=@true RequiredError="Password required." /><br />
```

To perform other validations, you can still use validation attributes by assigning them directly to the **Validation** property. Make sure you use **For** property as well to ensure that the error subject is the data field and not input component to construct the error message correctly.

Using validation attributes

```
<MudTextField Label="PASSWORD"
    InputType="InputType.Password" @bind-Value=Password
    Validation=@(new PasswordAttribute()) For=@(()=>Password)
    Required=@true RequiredError="Password required." /><br />
```

You can also assign a delegate to **Validation** property to call the method to perform validation. If validation is successful return **null**, if validation fails return a **string** that represents the error message. You can use **IEnumerable<string>** when returning multiple error messages using a **string** array or a **List<string>** object. The method can be synchronous or asynchronous.

Let's say that we accept either a user Id or email in the login form and we only want to validate it as email if it has a @ character. We will create **EmailAddress** validation attribute and then add a **IsValidUserId** method that calls the validation attribute to validate only if the input value contains the @ character. If the input value is not valid we will use the validation attribute to construct the error message.

Using method to perform conditional validation

```
private EmailAddressAttribute EmailValidator = new();

string IsValidUserId(string value) {
    if (!value.Contains('@') || EmailValidator.IsValid(value)) return null;
    return EmailValidator.FormatErrorMessage("User id/email");
}
```

Assigning delegate to validation method

```
<MudTextField Label="USER ID/EMAIL"
    InputType="InputType.Email" @bind-Value=UserName
    Required=@true RequiredError="User ID or Email required."
    Validation=@(new Func<string,string>(IsValidUserId)) /><br />
```

Usually for text input components, validation only occurs on lost focus since the value is only updated at **onBlur** event and not **onChange** event. Add **Immediate** property on the component to update the value the moment the text is changed.

Immediate updating of value

```
<MudTextField Label="USER ID/EMAIL" Immediate
    InputType="InputType.Email" @bind-Value=UserName
    Required=@true RequiredError="User ID or Email required."
    Validation=@(new Func<string,string>(IsValidUserId)) /><br />
<MudTextField Label="PASSWORD" Immediate
    InputType="InputType.Password" @bind-Value=Password
    For=@(()=>Password) Validation=@(new PasswordAttribute())
    Required=@true RequiredError="Password required." /><br />
```

In the **Events** page, notice that filtering does not begin until you press enter or move away from the search input field. Add **Immediate** property to filter upon text change. Use this option with care as this can impact input performance if you are performing time consuming operations everytime a value is updated.

Filtering on text change: Pages\Events.razor

```
<MudTextField @bind-Value="SearchText" Placeholder="Search"
    Adornment=@Adornment.Start AdornmentIcon=@Icons.Material.Filled.Search
    IconSize=@Size.Medium Immediate />
```

Also note while database searches can be case-insensitive but not string comparisons performed in .NET code so you will need to pass in a **StringComparison** instead as well. Replace the **Filter** in the **MudTable** with the following expression.

Case-insensitive search

```
string.Concat(item.Id,item.Type,item.Title).Contains(SearchText,
    StringComparison.InvariantCultureIgnoreCase)
```

2 MudBlazor Services

2.1 Toast Notifications

Toast notifications are popup content that stay on screen for a certain amount of time and will disappear on their own. Since they are usually rendered on another layer that is independent and on top of other page content, they remain visible even though the page content changes. In MudBlazor, such a toast notification is know as a **Snackbar**. Make sure to call **AddMudServices** on the application host builder to have access to this service.

Add MudBlazor services

```
builder.Services.AddMudServices();
```

Add a **MudSnackbarProvider** to your layout to setup the UI for the notifications to appear. This guarantees that snackbars would appear no matter what is the page and components currently being rendered.

Adding Snackbar component to UI

```
@inherits LayoutComponentBase

<MudThemeProvider />
<MudSnackbarProvider />
<MudLayout>
    :
```

Whichever page or component that needs to display a toast notification, you can get access to the **Snackbar** service through property injection of **ISnackbar**. Call **Add** method to add a snackbar notification.

Accessing the Snackbar service: Pages\events\Create.razor

```
@inject ISnackbar Snackbar
```

Adding a snackbar notification

```
async void HandleSubmitAsync(EditContext context) {
    try {
        Success = string.Empty;
        Failure = string.Empty;
        ValidationFailed = false;
        await DataService.AddEventAsync(Item);
        Success = $"Event {Item.Id} added successfully";
        Snackbar.Add(Success, Severity.Success);
        Item = new Event { From = DateTime.Now };
    }
                :
```

The location and other options of the **Snackbar** can be configured when you call the **AddMudServices** method.

Configuring Snackbar: Program.cs

```
builder.Services.AddMudServices(config => {
    config.SnackbarConfiguration.PositionClass = Defaults.Classes.Position.TopRight;
    config.SnackbarConfiguration.PreventDuplicates = false;
    config.SnackbarConfiguration.NewestOnTop = false;
    config.SnackbarConfiguration.ShowCloseIcon = true;
    config.SnackbarConfiguration.VisibleStateDuration = 10000;
    config.SnackbarConfiguration.HideTransitionDuration = 500;
    config.SnackbarConfiguration.ShowTransitionDuration = 500;
    config.SnackbarConfiguration.SnackbarVariant = Variant.Filled;
});
```

You can also configure options per snackbar notification by providing a delegate to set the options for the new notification. The following options below will close a snackbar if you navigate to another page and the snackbar has no close icon button.

Snackbar configuration per notification

```
Snackbar.Add(Success, Severity.Success, options => {
    options.CloseAfterNavigation = true;
    options.ShowCloseIcon = false;
});
```

Sometimes the purpose of using the snackbar is to display messages as we navigate between pages. For this you should not set **CloseAfterNavigation** to **true**.

By default a Snackbar can stay even if we navigate to another page

```
async Task HandleSubmitAsync() {
    try {
        IsSubmitting = true;
        Error = string.Empty;
        await ParticipantDataService.AddParticipantAsync(Item);
        Snackbar.Add($"User '{Item.Id}' registered.", Severity.Success);
        Navigation.NavigateTo("/user/login");
                :
```

2.2 Dialogs

Since screen space is limited, we can use popups to display additional details and to perform additional operations while remaining on the same page. To use dialogs, add **MudDialogProvider** into your layout.

Adding dialog support component to UI

```
@inherits LayoutComponentBase

<MudThemeProvider />
<MudSnackbarProvider />
<MudDialogProvider />
<MudLayout>
```

A dialog is implemented as a separate component. Add a **EventDetails** component to the **Shared** folder. Use the **MudDialog** component to encapsulate the content of the dialog. The dialog can contain different section to construct the title, the main content and action area. Since the actual dialog will be instantiated by the dialog service and provided as a **CascadingValue**, you need to define a property to capture and access it. A cascading value is automatically passed from ancestor to all descendants. To get this value, you need to use mark your property using **CascadingParameter** attribute and Blazor will match the value to the property based on the type. Use **Parameter** on additional properties that will be passed directly during instantiation.

A basic MudBlazor dialog: Shared\EventDetails.razor

```
<MudDialog>
    <TitleContent></TitleContent>
    <DialogContent></DialogContent>
    <DialogActions></DialogActions>
</MudDialog>

@code {
    [CascadingParameter]MudDialogInstance MudDialog { get; set; }
    [Parameter]public Event Event { get; set; }
}
```

The dialog title can be passed in during instantiation, this is useful if the title changes everytime you open the dialog. If you want a fixed or more complex title, you can add a **TitleContent** section. Here we conditionally use a **MudChip** to tag an online course and a **MudDivider** to create a separator line.

TitleContent section

```
<TitleContent>
    <MudText Typo=@Typo.h5>Event Details</MudText>
    <MudDivider />
</TitleContent>
```

In the main content area, we will display the other details of the event. Optional fields like **Description** will only be displayed if it's not empty. Use **FullWidth** property to ensure a particular input gets the most space with other inputs on the same row.

DialogContent section

```
<MudStack Row=@true>
    <MudSpacer />
    @if (Event.Online) { <MudChip Color=@Color.Info>ONLINE</MudChip> }
</MudStack>

<MudStack Row=@true>
    <MudTextField Label="ID" Value=@Event.Id ReadOnly />
    <MudTextField FullWidth Label="TITLE" Value=@Event.Title ReadOnly />
</MudStack><br />

<MudStack Row=@true>
    <MudTextField Label="TYPE" Value=@Event.Type ReadOnly />
    <MudTextField Label="FROM" Value=@Event.From ReadOnly />
    <MudTextField Label="HOURS" Value=@Event.Hours ReadOnly />
</MudStack><br />
```

```
@if (!string.IsNullOrEmpty(Event.Description)) {
    <MudTextField Label="DESCRIPTION"
        Value=@Event.Description Lines="5" ReadOnly />
    <br />
}

<MudStack Row=@true>
    <MudTextField Label="SEATS" Value=@Event.Seats ReadOnly />
    <MudTextField Label="ALLOCATED SEATS" Value=@Event.AllocatedSeats ReadOnly />
    <MudTextField Label="REMAINING SEATS" Value=@Event.RemainingSeats ReadOnly />
</MudStack><br />

@if (Event.Fee != 0) {
    <MudTextField Label="FEE"
      Value=@Event.Fee ReadOnly /><br />
}
```

Dialogs are also commonly used for input. The above content can be easily change for input by wrapping the content in a **MudForm** or **EditForm** and removing **ReadOnly** property or binding it to a state property so you can easily toggle between view and edit mode. Also make sure you are using a two-way binding for all input components; for example **@bind-Value** instead of **Value**.

Example of using state property to toggle editing

```
bool IsEditing { get; set; }
```

Example of input component configured for both view and edit mode

```
<MudTextField FullWidth Label="TITLE" @bind-Value=Event.Title
    Required RequiredError = "Title is required."
    ReadOnly=@(!IsEditing) />
```

We will now use the above dialog in the **Events** page. Use **IDialogService** to access the dialog service. Add a **ShowDetails** method to use the service to open a dialog for an **Event**. Call the generic **Show** method to instantiate and open the specific dialog, use a **DialogParameters** object to pass in additional parameters. If your dialog does not have a fixed title, pass in the title as the first argument to **Show** method.

Get access to dialog service: Pages\Events.razor

```
@page "/events"
@inject EventDataService DataService
@inject IDialogService DialogService
```

Instanting a dialog component with parameters

```
void ShowDetails(Event item) {
    var parameters = new DialogParameters();
    parameters.Add("Event", item);
    DialogService.Show<EventDetail>(null, parameters);
}
```

Inside a **MudTable** you can refer to the current item through **context** property. This would allow you to easily write code to call the above method and pass in the item. To do this, do not bind to the method but bind to method call statement. You can either add a link or icon button for each item in the table or wrap an existing column into a link. The following shows how to convert the Title column into a link to trigger the **ShowDetails** method and pass in the current **Event**.

Link to call the ShowDetails method

```
<RowTemplate>
    <MudTd DataLabel="ID">@context.Id</MudTd>
    <MudTd DataLabel="Type">@context.Type</MudTd>
    <MudTd DataLabel="Title">
        <MudLink OnClick=@(()=>ShowDetails(@context))>@context.Title</MudLink>
    </MudTd>
        :
```

Run and test out the above changes. By default clicking outside the dialog would close it. You can configure the dialog by passing in an additional **DialogOptions** object to the **Show** method as shown below. Use can show a close button, enable closing the dialog by pressing **Escape** key, disable closing on clicking outside a dialog, increase the width of the dialog to fit the content, or resize the dialog to take up the entire screen.

Configuring a dialog

```
void ShowDetails(Event item) {
    var parameters = new DialogParameters();
    parameters.Add("Event", item);
    var options = new DialogOptions {
        CloseButton = true,
        CloseOnEscapeKey = true,
        DisableBackdropClick = true,
        //  FullWidth = true,
        //  FullScreen = true
    };
    DialogService.Show<EventDetail>(null, parameters, options);
}
```

Rather than only configuring per dialog, you can configure these options for all dialogs by setting the properties directly on the **MudDialogProvider** on the layout. They will become the default settings for **DialogOptions**. You can still override the settings per dialog.

Setting default options for all dialogs: MainLayout.razor

```
<MudDialogProvider CloseButton=@true CloseOnEscapeKey=@true FullWidth=@true />
```

When a dialog is used to change data it may require to inform its parent component if the data is updated. The data may be used in the rendering of the parent component so it needs to be refreshed. For an *admin* user, we will allow the dialog to be used to remove the **Event**. We can use a delegate parameter that can be assigned a method in the parent component that will refresh its own state. Once the item is removed, we can then use the delegate to call the method. You can close the dialog from code as you have access to the dialog through the **MudDialog** property.

Get required services: Shared\EventDetails.razor

```
@inject EventDataService EventDataService
@inject ISnackbar Snackbar
```

A delegate parameter and state property for processing

```
[Parameter]public Action<Event> EventUpdated { get; set; }
bool IsProcessing { get; set; }
```

Method to delete the event

```
async Task DeleteEventAsync() {
    try {
        IsProcessing = true;
        await EventDataService.RemoveEventAsync(Event.Id);
        Snackbar.Add($"Event '{Event.Id}' deleted.", Severity.Success);
        EventUpdated?.Invoke(Event);
    }
    catch (Exception) { Snackbar.Add($"Cannot delete event '{Event.Id}'.", Severity.Error); }
    finally { MudDialog.Close(); }
}
```

Add button to DialogActions section

```
@if (IsProcessing) {
    <MudProgressCircular Class="mx-auto" Color=@Color.Primary Indeterminate=@true /> }
else {
    <AuthorizeView Roles="admin">
        <MudButton Class="ml-auto" Color=@Color.Error Variant=@Variant.Filled
            OnClick=@DeleteEventAsync>DELETE</MudButton>
    </AuthorizeView>
}
```

The parent component provides a method and passes a delegate to the method as a parameter to the dialog. The method reloads the data and this should update the UI. However, sometimes dynamically generated content may not refresh properly so call the **StateHasChanged** method to force a UI refresh.

Method to reload Event list and refresh page state

```
async void OnEventUpdated(Event item) {
    Items = await DataService.GetEventsAsync();
    StateHasChanged();
}
```

Passing delegate as a parameter to a component

```
void ShowDetails(Event item) {
    var parameters = new DialogParameters();
    parameters.Add("Event", item);
    parameters.Add("EventUpdated", new Action<Event>(OnEventUpdated));
    var options = new DialogOptions { CloseButton = false };
    DialogService.Show<EventDetail>(null, parameters, options);
}
```

We want to add more features to the **EventDetails** component. We want a user to be able to enter and remove their participation in an event. We also need to check if the user is already participating in an event. You can go about this in two ways; locate the event and access the participants for that event or locate the participant instead and access the events for that participant. In Entity Framework use **Include** method when you wish to include related entities to a particular entity you are retrieving. The following shows how to retrieve an **Event** together with related **Participants**. We can then check for a particular participant of the event.

Check an event for a particular participant: Services\EventDataService.cs

```
public async Task<bool> HasParticipantAsync(string eventId, string participantId) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var eventItem = await dc.Events.Include(e => e.Participants).SingleOrDefaultAsync(
            e => e.Id == eventId);
        if (eventItem != null) return eventItem.Participants.SingleOrDefault(
            p => p.Id == participantId) != null;
        return false;
    }
}
```

Adding and removing a participant from an event can be quite an involved task. This is where creating stored procedures may be a better option instead of coding in C#. A database transaction is required since we need to ensure that no one else can update or access **AllocatedSeats** column in **Events** table until the end of the transaction by using the **RepeatableRead** isolation level.

Adding participant to event

```
public async Task AddParticipantAsync(string eventId, string participantId) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var transaction = await dc.Database.BeginTransactionAsync(
            IsolationLevel.RepeatableRead);
        try {
            var eventItem = await dc.Events.Include(e => e.Participants)
                .SingleOrDefaultAsync(e => e.Id == eventId);
            if (eventItem == null) throw new Exception($"Event '{eventId}' does not exist.");
            if (eventItem.Participants.SingleOrDefault(p => p.Id == participantId) != null)
                throw new Exception($"User '{participantId}' is already a participant of event
'{eventId}'.");

            if (eventItem.RemainingSeats == 0)
                throw new Exception($"Event '{eventId}' is full.");
            var participant = await dc.Participants.FindAsync(participantId);
            if (participant == null) throw new Exception(
                $"Participant '{participantId}' does not exist.");
            eventItem.Participants.Add(participant);
            eventItem.AllocatedSeats++;
            await dc.SaveChangesAsync();
            await transaction.CommitAsync();
        }
        catch (Exception) { await transaction.RollbackAsync(); throw; }
    }
}
```

Removing participant from event

```
public async Task RemoveParticipantAsync(string eventId, string participantId) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var transaction = await dc.Database.BeginTransactionAsync(
            IsolationLevel.RepeatableRead);
        try {
            var eventItem = await dc.Events.Include(e => e.Participants)
                .SingleOrDefaultAsync(e => e.Id == eventId);
            if (eventItem == null) throw new Exception($"Event '{eventId}' does not exist.");
            var participant = eventItem.Participants.SingleOrDefault(
                p => p.Id == participantId);
            if (participant == null) throw new Exception(
                $"User '{participantId}' is not a participant of event '{eventId}.");
            eventItem.Participants.Remove(participant);
            eventItem.AllocatedSeats--;
            await dc.SaveChangesAsync();
            await transaction.CommitAsync();
        }
        catch (Exception) {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```

In **EventDetails** we need to access the **ClaimsPrincipal** from authentication state as we need to get the user Id to map the current user to a participant. From this we will find out if the user is participating in the current event and set **IsParticipating** state to reflect this. We will cache the principal in **User** property even though we only need the user Id.

Getting user information and event participation

```
@using System.Security.Claims;
@inject EventDataService EventDataService
@inject AuthenticationStateProvider AuthenticationStateProvider
@inject ISnackbar Snackbar
            :
@code {
            :
    ClaimsPrincipal User { get; set; }
    bool IsParticipating { get; set; }

    protected override async Task OnParametersSetAsync() {
        User = (await AuthenticationStateProvider.GetAuthenticationStateAsync()).User;
        if (User.Identity.IsAuthenticated) {
            IsParticipating = await EventDataService.HasParticipantAsync(
                Event.Id, User.Identity.Name);
        }
    }
}
```

We will now add two methods; **ParticipateEventAsync** and **LeaveEventAsync** that will add and remove the current user as the participant in the current event based on the event Id retrieve from **Event** object and user Id retrieved from **User** object.

Method to add user as participant

```
async Task ParticipateEventAsync() {
    try {
        IsProcessing = true;
        await EventDataService.AddParticipantAsync(Event.Id, User.Identity.Name);
        Snackbar.Add($"Participation in event '{Event.Id}' confirmed.", Severity.Success);
        IsParticipating = true; EventUpdated?.Invoke(Event);
    }
    catch (Exception ex) { Snackbar.Add(ex.Message, Severity.Error); }
    finally { MudDialog.Close(); }
}
```

Method to remove user as participant

```
async Task LeaveEventAsync() {
    try {
        IsProcessing = true;
        await EventDataService.RemoveParticipantAsync(Event.Id, User.Identity.Name);
        Snackbar.Add($"Your '{Event.Id}' event participation is cancelled.", Severity.Success);
        IsParticipating = false; EventUpdated?.Invoke(Event);
    }
    catch (Exception ex) { Snackbar.Add(ex.Message, Severity.Error); }
    finally { MudDialog.Close(); }
}
```

We will now add in the additional buttons that will be available as long as the user is authenticated. Which button appears depends on the **IsParticipating** state. We will also add a tag to highlight event participation with an extra **MudChip**.

Add event participation buttons

```
@if (IsProcessing) {
    <MudProgressCircular Class="mx-auto" Color=@Color.Primary Indeterminate=@true /> }
else {
    <AuthorizeView>
        @if (IsParticipating) {
            <MudButton Color=@Color.Primary
               Variant=@Variant.Filled OnClick=@LeaveEventAsync>
                LEAVE EVENT
            </MudButton>
        }
        else if (Event.RemainingSeats > 0) {
            <MudButton Color=@Color.Primary
               Variant=@Variant.Filled OnClick=@ParticipateEventAsync>
                PARTICIPATE EVENT
            </MudButton>
        }
    </AuthorizeView>
            :
```

Add extra tags for participation info

```
<MudStack Row=@true>
    <MudSpacer />
    @if (Event.Online) { <MudChip Color=@Color.Info>ONLINE</MudChip>   }
    @if (Event.RemainingSeats == 0) { <MudChip Color=@Color.Info>FULL</MudChip> }
    @if (IsParticipating) { <MudChip Color=@Color.Secondary>PARTICIPATING</MudChip> }
</MudStack>
```

Currently we close the dialog everytime we complete the operation so we do not need to reset or refresh the UI state. If we choose to maintain the dialog we would need to make sure the UI state is refreshed property. In **EventDetails** dialog, we will need to make sure **IsParticipating** is updated, **IsProcessing** is reset and we need to fetch a new updated **Event**. Add the method to retrieve a specific **Event**.

Method to retrieve a specific event: Services\EventDataService.cs

```
public async Task<Event> GetEventAsync(string id) {
    using (var dc = _dcFactory.CreateDbContext()) {
        return await dc.Events.FindAsync(id);
    }
}
```

Update the finalization in the following two methods to fetch the updated **Event** and reset **IsProcessing** state. The UI will detect the changes and refresh automatically. If for some reason, some UI components shows prior state, call the **StateHasChanged** method to force a UI refresh.

Refresh dialog content

```
async Task ParticipateEventAsync() {
                :
    finally {
        Event = await EventDataService.GetEventAsync(Event.Id);
        IsProcessing = false;
    }
}

async Task LeaveEventAsync() {
                :
    finally {
        Event = await EventDataService.GetEventAsync(Event.Id);
        IsProcessing = false;
    }
}
```

Not only the current component needs to refresh but the parent component or page may need to refresh as well. We've already implemented a **EventUpdated** parameter but we are using it for all operations. You may add a separate delegate parameter for each operation or define an **enum** to indicate the reason and pass as a parameter to the delegate method. You can pass additional parameters including the source object affected.

Example enum

```
public enum UpdateReason { ItemAdded, ItemRemoved, ItemUpdated }
```

Example delegate parameter

```
[Parameter]Action<UpdateReason, Event> EventUpdated;
```

Example of calling delegate method with result state and parameter

```
EventUpdated?.Invoke(UpdateReason.ItemRemoved, Event);
```