Blazor

ASP520-A

Module 3

Implementing Forms & Using MudBlazor Services

1 View Models

1.1 Event Models

Models are not only used for data saved to and loaded from data storage. Models can be specifically designed to collect data from specific input forms and to output results in specific formats which are commonly called as view models. View models are built to help resolve problems with using plain data models for UI purposes.

One issue of using **Event** model class in MudBlazor is that it has no control to input a **DateTime** value. It has a **DatePicker** that can be binded to a nullable **DateTime** but the **From** property of **Event** is not nullable. Also the control will store the date but it resets the time as well. MudBlazor also has a **TimePicker** but bindable to a nullable **TimeSpan**. Datatype and behavior differences means that the **Form** property cannot be directly used in an input form.

We can of course enhance our data model so that it can also be used as a view model by adding additional features. Following are two properties that we can add to **Event** class specifically to support requirements of **DatePicker** and **TimePicker** and ensure we preserve the time when date is stored and preserve the date when time is stored. Eventually date and time data still accessed and stored in the **From** property. To stop database migration from adding columns to **Events** table for these properties assign **NotMappedAttribute** to them.

Properties added to model for UI support: Models\Event.cs

```
[NotMapped]
public DateTime? FromDate {
    get { return From; }
    set {
        if (value != null) {
            From = new DateTime(
                value.Value.Year, value.Value.Month, value.Value.Day,
                From.Hour, From.Minute, From.Second
            );
        }
    }
}
[NotMapped]
public TimeSpan? FromTime {
    get { return From.TimeOfDay; }
    set {
        if (value != null) {
            From = new DateTime(
                From.Year, From.Month, From.Day,
                value.Value.Hours, value.Value.Minutes, value.Value.Seconds);
        }
    }
}
```

Alternatively, if you do not wish to convert your data model to a view model, you can extend the view model from your data model. **FromDate** and **FromTime** properties will only exist in the view model. The view model will be used for UI while remaining fully compatible to all features and functionality implemented for **Event**.

Extend class for UI support: Models\NewEvent.cs

```
namespace Newrise.Shared.Models {
    public class NewEvent : Event {
        public DateTime? FromDate { ... }
        public TimeSpan? FromTime { ... }
    }
}
```

1.2 Participant Models

An application may wish to control what users can do or cannot do and what they can see or cannot see. This is called as authorization. To be able to authorize users in our application, we need authentication. Most applications use a ID and password system to authenticate their users. Some applications implement multi-factor authentication (MFA) where additional information like email address, phone numbers and other user secrets are required for authentication purposes. The data model may need changes to support the authentication methods implemented in the application.

We will update **Participant** class to store the password hash. Passwords should never be stored for security reasons so only users should know their passwords. A password hash is a one-way encrypted version of the plain-text password.

Update model to store hashed password: Models\Participant.cs

```
public class Participant {
            :
        [Required]
        [StringLength(256)]
        public string PasswordHash { get; set; }
            :
}
```

Update the database

```
Add-Migration AddAuthentication
Update-Database
```

Even though we never store the original password in the database but it is required in the user registration input form. In fact we actually have to store two input passwords for comparison to ensure the user has entered the correct password. We thus need a view model to enter the passwords. We can then use data annotations to validate the passwords.

Extending view model from data model: Models\NewParticipant.cs

```
public class NewParticipant : Participant {

}
```

Password property and annotations

```
[Required(ErrorMessage="{0} is required.")]
[StringLength(16, ErrorMessage = "{0} cannot be more than {1} characters.")]
public string Password { get; set; }
```

Use a **DisplayAttribute** to change the name of the property in error messages. Use a **CompareAttribute** to compare one property with another. These are used below for the **ConfirmPassword** property.

ConfirmPassword property and annotations

```
[Display(Name = "Confirm Password")]
[Required(ErrorMessage = "{0} is required.")]
[StringLength(16, ErrorMessage = "{0} cannot be more than {1} characters.")]
[Compare(nameof(Password), ErrorMessage = "Passwords does not match.")]
public string ConfirmPassword { get; set; }
```

The above data annotations does not guarantee a strong password. You can make use of **RegularExpressionAttribute** to validate a strong password. Construct a pattern by combining parts shown below to form the validation and then add the range of all acceptable characters followed by mininum and maximum number of characters.

Password regular expression validation parts

```
(?=.*[A-Z])         Must have at least one uppercase character
(?=.*[a-z])         Must have at least one lowercase 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$@$!%*#?&]{8,16}$
```

Add constant for password format pattern

```
public const string PasswordFormat =
    @"(?=.*[A-Z])^(?=.*[a-z])(?=.*\d)(?=.*[$@$!%*?&])[A-Za-z\d$@$!%*#?&]{8,16}$";
```

Use RegularExpressionAttribute with pattern for properties

```
            :
[RegularExpression(PasswordFormat, ErrorMessage = "{0} is not valid.")]
public string Password { get; set; }

            :
[RegularExpression(PasswordFormat, ErrorMessage = "{0} is not valid.")]
public string ConfirmPassword { get; set; }
```

1.3 Pure View Models

While models are not compulsory for input forms, they are required if you wish to use data annotations for input validation. A login form where you request for just the user ID and password may not require a model as the properties can be part of the login page but having and using a model is always beneficial.

A pure view model for login: Models\LoginInfo.cs

```
using System.ComponentModel.DataAnnotations;

namespace Newrise.Shared.Models {
    public class LoginInfo {
        [Display(Name = "ID/Email")]
        [Required(ErrorMessage = "{0} is required.")]
        [StringLength(254, ErrorMessage = "{0} cannot be more than {1} characters.")]
        public string UserID { get; set; }

        [Required(ErrorMessage = "{0} is required.")]
        [StringLength(16, ErrorMessage = "{0} cannot be more than {1} characters.")]
        [RegularExpression(NewParticipant.PasswordFormat, ErrorMessage = "{0} is invalid.")]
        public string Password {
            get; set;
        }
    }
}
```

2 Forms & Validation

2.1 Edit Form

Even though MudBlazor has a **MudForm** component, you need to use ASP.NET Core's built-in **EditForm** if you want to use data annotations to validate your model so you do not need to add specific validation components to your form. All you need to do is add a DataAnnotationsValidator to the form and it will generate the validation based on the model you assigned to **Model** property of the form.To use ASP.NET Core form and input components, you need to import the following namespace.

Adding namespaces for ASP.NET Core form components: _Imports.Razor

```
@using Microsoft.AspNetCore.Components.Forms
                :
```

Also assign a method to **OnValidSubmit** event to process the model that has passed validation and optionally assign a method to **OnInvalidSubmit** if you want to detect and process failed validations. In the form we will use a **MudCard** to contain the form contents in a **MudCardContent** section and action buttons in the **MudCardActions** section. We add two **MudAlert** components to display success and error messages in the page after the page title. These components will not be added if the messages are empty. To submit contents of the form, you need a **Submit** button which will trigger the form to validate input and call the methods that are assigned to **OnValidSubmit** and **OnInvalidSubmit** depending on whether form validation is successful or failed.

Setup a new page to add events: Pages\Event\Create.razor

```
@page "/events/create"
@inject EventDataService EventDataService
<PageTitle>@(Program.AppName + " - New Event")</PageTitle>
<EditForm Model=Item OnValidSubmit=HandleSubmit>
    <DataAnnotationsValidator />
    <MudGrid>
        <MudItem xs="12">
            <MudCard Class="ma-4 pa-4">
                <MudCardContent></MudCardContent>
                <MudCardActions></MudCardActions>
            </MudCard>
        </MudItem>
    </MudGrid>
</EditForm>
@code {
    string Success { get; set; } = string.Empty;
    string Failure { get; set; } = string.Empty;
    NewEvent Item { get; set; } = new Event { From = DateTime.Now }
    async void HandleSubmitAsync(EditContext context) {
        await EventDataService.AddEventAsync(Item);
    }
}
```

Initial contents of the card

```
<MudText Typo=Typo.h5>New Event</MudText><br />
@if (Success != string.Empty) {
    <MudAlert Severity=Severity.Success>@Success</MudAlert><br />
}
@if (Failure != string.Empty) {
    <MudAlert Severity=Severity.Error>@Failure</MudAlert><br />
}
```

Action buttons of the card

```
<MudButton Color=Color.Primary Variant=Variant.Filled
   ButtonType=ButtonType.Submit>SAVE</MudButton>
<MudButton Class="ml-2" Color=Color.Primary Variant=Variant.Filled
   Href="/events">VIEW EVENTS</MudButton>
```

We have yet to add any input components for our **Event** model. You can use either ASP.NET Core's built-in input components or the ones provided by MudBlazor. We will be using MudBlazor input components so we don't have to use a Javascript UI library and CSS to create and style a responsive UI.

Normally each input component will be on a separate row, you can use **MudStack** to combine multiple input components into one row. However you should make sure that those components would fit well across all size breakpoints or use the **MudGrid** and **MudItem** components instead.

We used three different input components to input **Id**, **Type** and **Online** properties of an **Event** object; **MudTextField**, **MudSelect** and **MudCheckBox**. The **For** property is to return the identity of the entity property that this input component is used for so that validation error messages can be attached to the correct component.

Using MudStack to combine components into one row

```
<MudStack Row="true">
    <MudTextField Label="ID *" @bind-Value=Item.Id MaxLength=6
        For=@(() => Item.Id) AutoFocus /><br />
    <MudSelect T=EventType Label="EVENT TYPE *" AnchorOrigin=Origin.BottomCenter
        @bind-Value=Item.Type For=@(()=>Item.Type)>
        <MudSelectItem T=EventType Value=EventType.Presentation />
        <MudSelectItem T=EventType Value=EventType.Training />
        <MudSelectItem T=EventType Value=EventType.Workshop />
        <MudSelectItem T=EventType Value=EventType.Forum />
    </MudSelect>
    <MudCheckBox Class="pt-4" Label="Available Online" @bind-Value=Item.Online />
</MudStack><br />
```

If you try to test the above page you will get an error. Some components may require other components to be included into the page for them to work. Any component that uses popups require a **MudPopoverProvider** to be present. This can be included into the layout instead of every page.

Including MudPopoverProvider service component: Layout\DefaultLayout.cs

```
@inherits LayoutComponentBase
<MudThemeProvider />
<MudPopoverProvider />
```

Input components for Title and Description

```
<MudTextField Label="TITLE *" @bind-Value=Item.Title MaxLength=50
    For=@(() => Item.Title) /><br />
<MudTextField Label="DESCRIPTION" @bind-Value=Item.Description MaxLength=1000
    For=@(()=>Item.Description) Lines=5 /><br />
```

Use **MudDatePicker** and **MudTimePicker** to input date and time. This is where the view model properties **FromDate** and **FromTime** is used to ensure date and time will be combined and stored into a single **From** property.

Input components for From using view model properties

```
<MudStack Row=true>
    <MudDatePicker Label="START DATE" @bind-Date=Item.FromDate />
    <MudTimePicker Label="START TIME" @bind-Time=Item.FromTime />
</MudStack><br />
```

Use **MudNumericField** to input or edit numerical type properties. You can use **Min** and **Max** to control the range of values so that you do not even need to use a **Range** validator. There are up and down buttons attached by default to adjust the value. You can set **HideSpinButtons** to **true** if you don't want the buttons to appear.

Rest of the input components

```
<MudStack Row=true>
    <MudNumericField Label="HOURS" @bind-Value=Item.Hours
        Min=0.5 Max=8.0 Step=1.0 For=@(()=>Item.Hours) />
    <MudNumericField Label="SEATS" @bind-Value=Item.Seats
        Min=1 Max=200 Step=10 For=@(()=>Item.Seats) />
    <MudNumericField Label="FEE" @bind-Value=Item.Fee
        Min=0 Max=5000 Step=100 For=@(()=>Item.Fee) />
</MudStack><br />
```

All input controls are now in the form but we need to update **HandleSubmitAsync** to ensure the entire process is handled correctly and properly. First we need to remove any previous messages displayed in the page by clearing **Success** and **Failure** then use the **EventDataService** to add the event to the database. We cannot guarantee it will succeed so the entire operation require at least a **try-catch** block to handle errors otherwise we display a success message and reset the form so another event can be added. Since this method is asynchronous, Blazor may not know when to update the page so call **StateHasChanged** method to inform Blazor to update UI changes in the component and these changes will be reflected to the web browser.

Method to process the form after validation

```
async void HandleSubmitAsync() {
    try {
        Success = string.Empty;
        Failure = string.Empty;
        await EventDataService.AddEventAsync(Item);
        Success = $"Event {Item.Id} added successfully";
        Item = new NewEvent { From = DateTime.Now };
    }
    catch (Exception ex) { Failure = $"{ex.Message} Cannot add event."; }
    finally { StateHasChanged(); }
}
```

2.2 Database Errors

We have exception handling in the above method because input validation would not guarantee that database operations will succeed. Operations may still fail due to table and column constraints. Entity Framework does not return the actual database error, it will return a general Entiry Framework error message and the actual error message can be retrieved from **InnerException** of the error. It is pointless to do so unless the error is produced from a stored procedure, otherwise a constraint error is often too technical to be understood by a normal user. A common error that we will get from the above operation is a primary key violation error where you try to add more than one event with the same **Id**. What we can do is to check for a duplicate event Id before adding the event. In this way, we can create a better error message that the one from the database server or from Entity Framework.

Avoiding constraint errors by checking in code: Services\EventDataService.cs

```
public async Task AddEventAsync(Event item) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        if (await dc.Events.FindAsync(item.Id) != null)
            throw new Exception($"Event with ID '{item.Id}' already exists.");
        await dc.Events.AddAsync(item);
        await dc.SaveChangesAsync();
    }
}
```

2.3 Validation Summary

Currently validation errors is attached to the input components which is good enough. Optionally you can also add in a **ValidationSummary** component to render all of the error messages. The following demonstrates showing the validation summary inside a **MudAlert** component controlled by a **ValidationFailed** property. The property can be set when validation fails which we can detect through a **OnInvalidSubmit** method call. We can simply use a lambda expression to trigger the state change. Remember to reset the property in your submit method.

ValidationSummary control property

```
bool ValidationFailed { get; set; }
```

Displaying validation summary

```
@if (ValidationFailed) {
        <MudAlert Severity="Severity.Error"><ValidationSummary /></MudAlert>
}
```

Triggering the above section when validation fails

```
<EditForm Model=Item
    OnInvalidSubmit=@(()=>ValidationFailed=true)
    OnValidSubmit=HandleSubmitAsync>
```

Reset the property on every submit

```
async void HandleSubmitAsync() {
    try {
        ValidationFailed = false;
                :
```

2.4 Input Focus

Notice that the input cursor is automatically positioned into the User ID input control when the page is opened. This is because we added an **AutoFocus** parameter to that control. We do not have to care if we are going to navigate away from the page after completing the operation but if we choose to remain in the same page, we may want to reset the focus back to that control but do this we need to have access to it. To be able to access components created in the page you need to declare a field or property for the component and use **ref** directive to bind the component to it.

Declare a field for a component

```
MudTextField<string> Id;
```

Bind component to the field

```
<MudTextField @ref=Id Label="ID *" @bind-Value=Item.Id MaxLength=6
    For=@(() => Item.Id) AutoFocus /><br />
```

Refocus on component

```
async void HandleSubmitAsync() {
    try { ... } catch (Exception ex) { ... }
    finally {
        await Id.FocusAsync();
        StateHasChanged();
    }
}
```

2.5 Page State

An issue with asynchronous operations is that the page is still fully interactable while those operations are in progress. This means that it is possible for the user to submit the form again even though we are already in the process of updating the database. To control this you can add a state field or property as shown below that can be used to determine whether the submission process is in progress.

Adding a page state property

```
bool IsSubmitting { get; set; }
```

Setting and reseting the state

```
async void HandleSubmitAsync() {
    try { IsSubmitting = true; ... }
    catch (Exception ex) { ... }
    finally { IsSubmitting = false; ... }
}
```

Based on the state you can choose to disable controls using the **Disabled** parameter. Below shows how to disable the actions buttons while submission is in progress. You can also choose to hide or replace anything in the page. This will stop the user from doing anything in the page until the operation has completed.

Disabling buttons

```
<MudButton Disabled=IsSubmitting Color=Color.Primary Variant=Variant.Filled
           ButtonType=ButtonType.Submit>SAVE</MudButton>
<MudButton Disabled=IsSubmitting Class="ml-2" Color=Color.Primary Variant=Variant.Filled
           Href="/events">VIEW EVENTS</MudButton>
```

Replacing form with progress alert

```
@if (IsSubmitting) {
    <MudAlert Severity=Severity.Info>
        <MudText>Attempting to add event. Please wait.</MudText>
        <MudProgressCircular Color=Color.Inherit Indeterminate />
    </MudAlert>
}
else {
    <MudCard Class="ma-4 pa-4"> ... </MudCard>
}
```

2.6 Using Tables

We can now update **Events** page to display the list of events added to the database. You can use **MudSimpleTable** or **MudTable**. Advantage of using the later is that you can assign items to be rendered and it generates one row for each item by using a **RowTemplate** without a **for-each** loop. It has support for paging, sorting, filtering, selection and editing features. It has many different sections like **ToolBarContent**, **HeaderContent**, **FooterContent** and **PagerContent** as well.

Displaying a list of events: Pages\Events.razor

```
@page "/events"
@using EventModel = Newrise.Shared.Models.Event;
@inject EventDataService EventDataService
<PageTitle>@(Program.AppName + " - Event Listing")</PageTitle>
<MudGrid>
    <MudItem Class="ma-8" xs="12">
        <MudText Typo="Typo.h5">List of Events</MudText><br />
        <MudTable Items=Items Bordered Striped RowsPerPage="10">
            <ToolBarContent></ToolBarContent>
            <HeaderContent></HeaderContent>
            <RowTemplate></RowTemplate>
            <PagerContent></PagerContent>
        </MudTable>
    </MudItem>
</MudGrid>
@code {
    List<EventModel> Items { get; set; } = new List<EventModel>();
    protected override async Task OnInitializedAsync() {
        Items = await EventDataService.GetEventsAsync();
    }
}
```

ToolBarContent section

```
<MudButton Variant=Variant.Filled Color=Color.Primary
    Href="/events/create">ADD EVENT</MudButton>
```

HeaderContent section

```
<MudTh>ID</MudTh>
<MudTh>Type</MudTh>
<MudTh>Title</MudTh>
<MudTh>From</MudTh>
<MudTh>Hours</MudTh>
<MudTh>Seats</MudTh>
<MudTh>Fee</MudTh>
```

RowTemplate section

```
<MudTd DataLabel="ID">@context.Id</MudTd>
<MudTd DataLabel="Type">@context.Type</MudTd>
<MudTd DataLabel="Title">@context.Title</MudTd>
<MudTd DataLabel="From">@context.From</MudTd>
<MudTd DataLabel="Hours">@context.Hours</MudTd>
<MudTd DataLabel="Seats">@context.Seats</MudTd>
<MudTd DataLabel="Fee">@context.Fee</MudTd>
```

PagerContent section

```
<MudTablePager />
```

Set **RowsPerPage** and add a **MudTablePager** to **PagerContent** to allow the user to change page and page options. To support filtering, add a string field or property to store the search text and add an input text field for the user to enter it, then assign a method to **Filter** property of **MudTable** to perform the filtering based on the search text.

Add a property to store search text

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

Add MudTextField to input the search text

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

Method to filter events

```
bool EventFilter(EventModel item) {
    return string
        .Concat(item.Id, item.Type, item.Title)
        .Contains(SearchText, StringComparison.InvariantCultureIgnoreCase);
}
```

Assign filter method to MudTable

```
<MudTable Items=Items Bordered Striped RowsPerPage="10" Filter=EventFilter>
```

By default **MudTextField** only stores the input value when it loses focus. This means validation and processing does not occur immediately during input. You need to press **ENTER** or move away from the search box for the filter to occur. Add **Immediate** to a **MudTextField** to store the value immediately during editing.

Enable Immediate option on MudTextField

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

To support sorting, add a **MudTableSortLabel** to any header column and assign the method to return the sorting value to the **SortBy** property. The default sort order is ascending but you can change it with the **InitialDirection** property.

Enable sorting on ID

```
<MudTh>
    <MudTableSortLabel
        SortBy=@(new Func<EventModel,object>(x => x.Id))>ID
    </MudTableSortLabel>
</MudTh>
```

Enable sorting on Title

```
<MudTh>
    <MudTableSortLabel
        SortBy=@(new Func<EventModel,object>(x => x.Title))>Title
    </MudTableSortLabel>
</MudTh>
```

Enable sorting on Fee

```
<MudTh>
    <MudTableSortLabel InitialDirection="SortDirection.Descending"
        SortBy=@(new Func<EventModel,object>(x => x.Fee))>Fee
    </MudTableSortLabel>
</MudTh>
```

3 Notifications & Dialogs

3.1 Notifications

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: Program.cs

```
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 service component: Layout\DefaultLayout.cs

```
@inherits LayoutComponentBase
<MudThemeProvider />
<MudPopoverProvider />
<MudSnackbarProvider />
        :
```

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\Event\Create.razor

```
@inject ISnackbar Snackbar
```

Adding a snackbar notification

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

3.2 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 />
<MudPopoverProvider />
<MudSnackbarProvider />
<MudDialogProvider />
        :
```

A dialog is implemented as a separate component. Add **EventDetails** component to a new **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 T=string 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 />
}
```

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 EventDataService
@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>
        :
```

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> EventDeleted { get; set; }
bool IsProcessing { get; set; }
```

Method to delete the event

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

Add button to DialogActions section

```
@if (IsProcessing) {
    <MudAlert Severity=Severity.Info>Deleting event. Please wait.</MudAlert> }
else {
    <MudSpacer />
    <MudButton Color=Color.Error Variant=Variant.Filled
        OnClick=DeleteEventAsync>DELETE</MudButton>
}
```

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 OnEventDeletedAsync(EventModel item) {
    Items = await EventDataService.GetEventsAsync();
    StateHasChanged();
}
```

Passing delegate as a parameter to a component

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