[Skip to content](https://chatgpt.com/c/68c1d4a8-9994-832d-b658-c19e915c2505#main)

**Chat history**

**You said:**

I have a MudButton where Variant is set to Variant.Outlined. Tell me how to toggle the variant to Variant.Filled in the mouse hover event

**ChatGPT said:**

MudBlazor components (like MudButton) don’t have a built-in OnMouseOver / OnMouseOut event like plain HTML. But you can achieve the hover toggle behavior by:

1. Wrapping the button inside a container that supports mouse events (e.g., a <div>).
2. Tracking the hover state in a boolean field.
3. Switching the Variant based on that boolean.

Here’s a clean Blazor example:

<div @onmouseover="() => isHovered = true" @onmouseout="() => isHovered = false"> <MudButton Variant="@(isHovered ? Variant.Filled : Variant.Outlined)" Color="Color.Primary"> Hover Me </MudButton> </div> @code { private bool isHovered; }

✅ Explanation:

* @onmouseover and @onmouseout come from Blazor’s built-in DOM events.
* When the mouse enters, isHovered becomes true → Variant.Filled.
* When the mouse leaves, isHovered becomes false → Variant.Outlined.

This is the simplest way in Blazor with MudBlazor to dynamically toggle styles/variants on hover.

Do you want me to also show you a **CSS-only solution** (no C# code-behind, purely styling override)?

**You said:**

Tell me how to refresh the page in MudBlazor

**ChatGPT said:**

In **MudBlazor**, there’s no direct API like MudBlazor.Refresh().  
Refreshing the page depends on whether you mean:

**🔹 1. Full Page Reload (like F5 / browser refresh)**

You can use Blazor’s NavigationManager to force a reload of the current page:

@inject NavigationManager Navigation <MudButton OnClick="RefreshPage" Color="Color.Primary" Variant="Variant.Filled"> Refresh Page </MudButton> @code { private void RefreshPage() { Navigation.NavigateTo(Navigation.Uri, forceLoad: true); } }

✅ This reloads the browser page completely, same as pressing F5.

**🔹 2. Re-render a Component (without full reload)**

If you only want the UI to update (not reload the entire page), you can call StateHasChanged() in your component:

<MudButton OnClick="ForceRender" Color="Color.Secondary" Variant="Variant.Outlined"> Refresh Component </MudButton> @code { private void ForceRender() { // Refresh this component only StateHasChanged(); } }

**🔹 3. Refresh Data (e.g., reload list or grid)**

If you’re refreshing data from a database or API, you should re-call the method that fetches the data and then trigger StateHasChanged(). Example with MudTable:

<MudTable Items="\_employees"> <ToolBarContent> <MudButton OnClick="LoadEmployees" Color="Color.Primary">Reload Data</MudButton> </ToolBarContent> </MudTable> @code { private List<Employee> \_employees = new(); protected override async Task OnInitializedAsync() { await LoadEmployees(); } private async Task LoadEmployees() { // Fetch fresh data (example only) \_employees = await EmployeeService.GetEmployeesAsync(); StateHasChanged(); } }

👉 Question for you:  
Do you want a **full browser reload** (like F5) or just a **data/UI refresh** inside the MudBlazor component?

**You said:**

I am developing an HR Saas system using Blazor WebApp. I am using MudBlazor for designing the UI. I have designed an employee search razor page where I used MudAutocomplete components to display the list of employee types and departments. Now, when I navigate to another page then comeback to the employee search page, it triggers the OnInitialized() event where it calls the web service to retrieve the data from the backend data. Tell me how to preserve the data for these MudAutocomplete components so that when navigating away from the page, it gets save and navigating back will restore that saved data and map it to these components.

**ChatGPT said:**

Got it 👍 You're running into the classic **state persistence** problem in Blazor:  
when you navigate away from a page, Blazor destroys that component instance. When you come back, OnInitialized() runs again and reloads data.

To make your MudAutocomplete keep its data between navigations, you need a **shared state** that lives longer than the page.

Here are three clean approaches (ranked from simplest to most robust for a SaaS system):

**🔹 1. Use a Scoped State Service (recommended)**

Create a service that holds the cached lists (EmployeeTypes, Departments). Register it as Scoped, so it lives for the user session (until logout or refresh).

**Step 1: Create a State Service**

public class EmployeeSearchState { public List<EmployeeTypeDto> EmployeeTypes { get; set; } = new(); public List<DepartmentDto> Departments { get; set; } = new(); public bool IsLoaded { get; set; } = false; }

**Step 2: Register in**Program.cs

builder.Services.AddScoped<EmployeeSearchState>();

**Step 3: Use in Your Page**

@page "/employee-search" @inject EmployeeSearchState State @inject IEmployeeService EmployeeService <MudAutocomplete T="EmployeeTypeDto" Label="Employee Type" ToStringFunc="e => e.Name" SearchFunc="@SearchEmployeeTypes" /> <MudAutocomplete T="DepartmentDto" Label="Department" ToStringFunc="d => d.Name" SearchFunc="@SearchDepartments" /> @code { protected override async Task OnInitializedAsync() { if (!State.IsLoaded) { State.EmployeeTypes = await EmployeeService.GetEmployeeTypesAsync(); State.Departments = await EmployeeService.GetDepartmentsAsync(); State.IsLoaded = true; } } private Task<IEnumerable<EmployeeTypeDto>> SearchEmployeeTypes(string value) { if (string.IsNullOrWhiteSpace(value)) return Task.FromResult(State.EmployeeTypes.AsEnumerable()); return Task.FromResult(State.EmployeeTypes .Where(x => x.Name.Contains(value, StringComparison.OrdinalIgnoreCase))); } private Task<IEnumerable<DepartmentDto>> SearchDepartments(string value) { if (string.IsNullOrWhiteSpace(value)) return Task.FromResult(State.Departments.AsEnumerable()); return Task.FromResult(State.Departments .Where(x => x.Name.Contains(value, StringComparison.OrdinalIgnoreCase))); } }

✅ Now the first time you visit the page, data loads from backend.  
✅ When you navigate away and return, Blazor reuses the cached data from EmployeeSearchState.

**🔹 2. Use a State Container with Persistent Cache**

If you want the data to survive **browser refresh** (F5), store it in **local storage or session storage** with [Blazored.LocalStorage](https://github.com/Blazored/LocalStorage).

* Cache EmployeeTypes and Departments in local storage the first time they’re loaded.
* On subsequent visits, read from local storage instead of calling backend.

**🔹 3. Use MudBlazor**MudAutocomplete**Virtualization + Lazy Loading**

If your employee types/departments are **very large lists**, you can skip caching and instead use server-side search (SearchFunc) with backend filtering. That way you don’t preload everything.

👉 Since you’re building an **HR SaaS app with Clean Architecture**, the best balance is:

* **Scoped service (**EmployeeSearchState**)** for session-level caching
* **Optional local storage** for persistence across refresh

Would you like me to extend this with a **Clean Architecture integration** (so the caching service lives in the Application layer, and UI only consumes it), instead of having state tied directly to the UI project?

**You said:**

Yes extend this with a Clean Architecture integration (so the caching service lives in the Application layer, and UI only consumes it), instead of having state tied directly to the UI project?

**ChatGPT said:**

Perfect 👍 — placing caching/state management in the **Application layer** keeps your **UI clean** and avoids re-fetching lookup/reference data like EmployeeTypes or Departments on every navigation.

Here’s how you can structure it in **Clean Architecture** for your HR SaaS system:

**🔹 1. Application Layer**

**a) Define a Lookup Cache Service Contract**

*// Application/Common/Interfaces/ILookupCacheService.cs* using System.Collections.Generic; using System.Threading.Tasks; public interface ILookupCacheService { Task<IReadOnlyList<EmployeeTypeDto>> GetEmployeeTypesAsync(); Task<IReadOnlyList<DepartmentDto>> GetDepartmentsAsync(); void ClearCache(); *// optional, e.g. after admin updates lookup data* }

**b) Implement the Service in Application**

*// Application/Common/Services/LookupCacheService.cs* public class LookupCacheService : ILookupCacheService { private readonly IEmployeeService \_employeeService; *// from Infrastructure or Domain service* private readonly IDepartmentService \_departmentService; private List<EmployeeTypeDto> \_employeeTypes; private List<DepartmentDto> \_departments; public LookupCacheService(IEmployeeService employeeService, IDepartmentService departmentService) { \_employeeService = employeeService; \_departmentService = departmentService; } public async Task<IReadOnlyList<EmployeeTypeDto>> GetEmployeeTypesAsync() { if (\_employeeTypes == null || !\_employeeTypes.Any()) { \_employeeTypes = await \_employeeService.GetEmployeeTypesAsync(); } return \_employeeTypes; } public async Task<IReadOnlyList<DepartmentDto>> GetDepartmentsAsync() { if (\_departments == null || !\_departments.Any()) { \_departments = await \_departmentService.GetDepartmentsAsync(); } return \_departments; } public void ClearCache() { \_employeeTypes = null; \_departments = null; } }

➡️ This ensures the **first call** loads from backend, but subsequent calls are served from memory.

**🔹 2. Infrastructure Layer**

You’ll already have **services/repositories** that hit your database (via EF Core or API).  
For example:

public class EmployeeService : IEmployeeService { private readonly IApplicationDbContext \_context; public EmployeeService(IApplicationDbContext context) { \_context = context; } public async Task<List<EmployeeTypeDto>> GetEmployeeTypesAsync() { return await \_context.EmployeeTypes .Select(e => new EmployeeTypeDto { Id = e.Id, Name = e.Name }) .ToListAsync(); } }

Same for DepartmentService.

**🔹 3. UI (Blazor WebApp)**

**a) Inject the Cache Service**

@page "/employee-search" @inject ILookupCacheService LookupCache <MudAutocomplete T="EmployeeTypeDto" Label="Employee Type" ToStringFunc="e => e.Name" SearchFunc="@SearchEmployeeTypes" /> <MudAutocomplete T="DepartmentDto" Label="Department" ToStringFunc="d => d.Name" SearchFunc="@SearchDepartments" /> @code { private IReadOnlyList<EmployeeTypeDto> \_employeeTypes = new List<EmployeeTypeDto>(); private IReadOnlyList<DepartmentDto> \_departments = new List<DepartmentDto>(); protected override async Task OnInitializedAsync() { \_employeeTypes = await LookupCache.GetEmployeeTypesAsync(); \_departments = await LookupCache.GetDepartmentsAsync(); } private Task<IEnumerable<EmployeeTypeDto>> SearchEmployeeTypes(string value) { if (string.IsNullOrWhiteSpace(value)) return Task.FromResult(\_employeeTypes.AsEnumerable()); return Task.FromResult(\_employeeTypes .Where(x => x.Name.Contains(value, StringComparison.OrdinalIgnoreCase))); } private Task<IEnumerable<DepartmentDto>> SearchDepartments(string value) { if (string.IsNullOrWhiteSpace(value)) return Task.FromResult(\_departments.AsEnumerable()); return Task.FromResult(\_departments .Where(x => x.Name.Contains(value, StringComparison.OrdinalIgnoreCase))); } }

**🔹 4. Register in DI (Startup)**

builder.Services.AddScoped<ILookupCacheService, LookupCacheService>();

**🔹 5. Optional Enhancements**

* ✅ **Expiration**: Add a timestamp in LookupCacheService to invalidate cache after X minutes.
* ✅ **LocalStorage fallback** (Blazored.LocalStorage) if you want persistence even after browser refresh.
* ✅ **ClearCache()** after **admin changes** to EmployeeTypes or Departments, so the cache is refreshed next time.

**🔑 Benefits**

* Application-layer owns lookup caching, not the UI.
* UI stays simple → just consumes ILookupCacheService.
* Future-proof: you can replace in-memory cache with Redis, distributed cache, or local storage without touching UI code.

Would you like me to also extend this so the **cache has an expiration policy** (like a lazy cache that reloads every 10 minutes), or is a simple in-memory "load once per session" good enough for your HR SaaS case?

**You said:**

I have put an EditForm component in my Employee.razor page as follow: <EditForm Model="@employee" OnValidSubmit="OnValidSubmit"> <MudPaper Class="pa-4 mud-height-full"> <MudText Typo="Typo.subtitle2">Validation Summary</MudText> @if (\_success) { <MudText Color="Color.Success">Success</MudText> } else { <MudText Color="@Color.Error"> <ValidationSummary /> </MudText> } </MudPaper> </EditForm> However, upon running the application, the following error occurred: The child content element 'CellTemplate' of component 'TemplateColumn' uses the same parameter name ('context') as enclosing child content element 'ChildContent' of component 'EditForm'. Specify the parameter name like: '<CellTemplate Context="another\_name"> to resolve the ambiguity Tell me how to fix this issue.

**ChatGPT said:**

Ah 👍 I see what’s happening.

That error:

The child content element 'CellTemplate' of component 'TemplateColumn' uses the same parameter name ('context')

as enclosing child content element 'ChildContent' of component 'EditForm'.

is a **naming collision**.

Blazor gives the default name context to the implicit variable in any child-content template (like ChildContent, CellTemplate, RowTemplate, etc.).  
Since your <EditForm> also has a child-content block (ChildContent), its implicit context is clashing with the CellTemplate inside your grid/table.

**✅ Fix: Rename the**Context**in your templates**

Whenever you see CellTemplate (or any other template inside a templated component), give it a different Context parameter name.

Example with a MudBlazor table:

<MudTable Items="@employees"> <HeaderContent> <MudTh>Name</MudTh> <MudTh>Department</MudTh> </HeaderContent> <RowTemplate Context="emp"> <MudTd DataLabel="Name">@emp.Name</MudTd> <MudTd DataLabel="Department">@emp.Department</MudTd> </RowTemplate> </MudTable>

Notice how I added Context="emp".  
Now, the implicit variable inside RowTemplate is emp, **not** context.  
That resolves the ambiguity with the EditForm.

**🔧 Applying to your case**

Inside your Employee.razor, find where you have a TemplateColumn or RowTemplate (likely in a MudTable or MudDataGrid) and rename the Context:

<TemplateColumn Title="Actions"> <CellTemplate Context="row"> <MudButton OnClick="@(() => EditEmployee(row))">Edit</MudButton> </CellTemplate> </TemplateColumn>

Instead of leaving it at the default Context="context".

**📝 Summary**

* Every templated component (EditForm.ChildContent, MudTable.RowTemplate, TemplateColumn.CellTemplate) has an implicit variable named context.
* When you nest them, you must **rename the inner one** to avoid clashes.
* Use Context="row", Context="emp", or any meaningful name instead of leaving it default.

👉 Do you want me to go through your **full**Employee.razor**page** layout and point out exactly where the Context should be renamed, based on MudBlazor MudTable + EditForm usage?

**You said:**

go through your full Employee.razor page layout and point out exactly where the Context should be renamed, based on MudBlazor MudTable + EditForm usage

**ChatGPT said:**

Great — let’s walk through this carefully.  
Your error means you have an EditForm **and** a MudTable (or MudDataGrid) on the same page. Both use templated content, and both default to the implicit variable name context.

When Blazor compiles, it sees multiple nested context variables → **collision**.

**Typical**Employee.razor**Layout (what you likely have)**

<EditForm Model="@employee" OnValidSubmit="OnValidSubmit"> <MudPaper Class="pa-4 mud-height-full"> <MudText Typo="Typo.subtitle2">Validation Summary</MudText> @if (\_success) { <MudText Color="Color.Success">Success</MudText> } else { <MudText Color="@Color.Error"> <ValidationSummary /> </MudText> } <MudTable Items="@employees"> <HeaderContent> <MudTh>Name</MudTh> <MudTh>Department</MudTh> <MudTh>Actions</MudTh> </HeaderContent> <!-- 🔴 PROBLEM HERE: default Context="context" --> <RowTemplate> <MudTd DataLabel="Name">@context.Name</MudTd> <MudTd DataLabel="Department">@context.Department</MudTd> <MudTd> <MudButton OnClick="@(() => EditEmployee(context))">Edit</MudButton> </MudTd> </RowTemplate> </MudTable> </MudPaper> </EditForm>

⚠️ The RowTemplate uses the default context, which collides with the implicit context from <EditForm>.

**✅ Fixed Version — Rename the**Context

<EditForm Model="@employee" OnValidSubmit="OnValidSubmit"> <MudPaper Class="pa-4 mud-height-full"> <MudText Typo="Typo.subtitle2">Validation Summary</MudText> @if (\_success) { <MudText Color="Color.Success">Success</MudText> } else { <MudText Color="@Color.Error"> <ValidationSummary /> </MudText> } <MudTable Items="@employees"> <HeaderContent> <MudTh>Name</MudTh> <MudTh>Department</MudTh> <MudTh>Actions</MudTh> </HeaderContent> <!-- 🟢 FIX: give RowTemplate its own Context --> <RowTemplate Context="emp"> <MudTd DataLabel="Name">@emp.Name</MudTd> <MudTd DataLabel="Department">@emp.Department</MudTd> <MudTd> <MudButton OnClick="@(() => EditEmployee(emp))">Edit</MudButton> </MudTd> </RowTemplate> </MudTable> </MudPaper> </EditForm>

**🔑 Rule of Thumb**

* <EditForm> uses ChildContent → its implicit parameter is context.
* <RowTemplate>, <CellTemplate>, <PagerContent>, etc., also use context.
* Always rename inner templates:
  + Context="emp" for employees
  + Context="dept" for departments
  + Context="row" for generic tables

👉 Do you also have a TemplateColumn**inside a**MudDataGrid? If so, I can show you the same Context fix specifically for CellTemplate. Would you like me to include that too?

**You said:**

I have added a Label attribute to my EmployeeDTO class but an error has occured during compilation. The error says: The type of namespace "LabelAttribute" could not be found (are you missing a using directive or an assembly reference. Note that I have added the following using directive in the C# code behind: using System.ComponentModel.DataAnnotations; Tell me how to fix this issue.

**ChatGPT said:**

Ah, I see what’s happening 👇

LabelAttribute **does not exist** in System.ComponentModel.DataAnnotations.  
That’s why you’re getting:

The type or namespace name 'LabelAttribute' could not be found

**✅ Why**

* Data Annotations in .NET include attributes like [Required], [StringLength], [Display], [Key], etc.
* There is **no built-in**[Label]**attribute**.

So if you want to display a friendly label for your DTO property, you should use [Display] instead.

**✅ Fix: Use**[Display(Name="...")]

Example:

using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Key] public int Id { get; set; } [Display(Name = "Employee Name")] public string Name { get; set; } [Display(Name = "Department")] public string Department { get; set; } }

Now, Blazor + MudBlazor components (like MudForm, MudTextField For=...) will pick up Display(Name="...") and render it as the label.

**🔹 If you *really* want**[Label]

You’d have to define a **custom attribute** yourself:

[AttributeUsage(AttributeTargets.Property, Inherited = false, AllowMultiple = false)] sealed class LabelAttribute : Attribute { public string Text { get; } public LabelAttribute(string text) => Text = text; }

Then:

public class EmployeeDto { [Label("Employee Name")] public string Name { get; set; } }

But MudBlazor (and Blazor validation) won’t recognize [Label] automatically — you’d need to write reflection logic to read it.

👉 Since you’re using **MudBlazor** and EditForm, the best approach is to stick with [Display(Name="...")].

Do you want me to also show you how to bind that [Display(Name="...")] automatically to a MudTextField label in your EditForm?

**You said:**

show me how to bind that [Display(Name="...")] automatically to a MudTextField label in the EditForm

**ChatGPT said:**

Perfect 👍 — MudBlazor plays really nicely with **DataAnnotations**.  
If you decorate your DTO properties with [Display(Name="...")], you don’t need to manually hardcode labels — MudTextField can pick them up.

**✅ Example: DTO with**[Display]

using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required] [Display(Name = "Employee Name")] public string Name { get; set; } [Display(Name = "Department")] public string Department { get; set; } [Display(Name = "Email Address")] [EmailAddress] public string Email { get; set; } }

**✅ Blazor Page (**Employee.razor**)**

@page "/employee" @using System.ComponentModel.DataAnnotations @inject NavigationManager Navigation <EditForm Model="@employee" OnValidSubmit="OnValidSubmit"> <DataAnnotationsValidator /> <MudPaper Class="pa-4"> <MudTextField For="@(() => employee.Name)" /> <MudTextField For="@(() => employee.Department)" /> <MudTextField For="@(() => employee.Email)" /> <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Save </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private void OnValidSubmit() { // Save logic here } }

**🔑 What’s Happening**

* MudTextField has a special property For that accepts a lambda expression (() => employee.Name).
* When you use For, MudBlazor looks at the property’s **DataAnnotations**.
* If the property has [Display(Name="...")], MudBlazor automatically uses that for the Label of the field.
* If you also add [Required], [EmailAddress], [StringLength], etc., those validations automatically plug into the <EditForm> via <DataAnnotationsValidator />.

So with the above, your form will render:

* Label **Employee Name** for the first field
* Label **Department** for the second
* Label **Email Address** for the third

Without you hardcoding Label="...".

⚡ This is the “Blazor way” — strongly typed + automatic annotation binding.

👉 Do you want me to also show how to **loop through all properties of EmployeeDto dynamically** (reflection) and generate fields with their [Display(Name)], so you don’t even write each MudTextField manually?

**You said:**

show how to loop through all properties of EmployeeDto dynamically (reflection) and generate fields with their [Display(Name)], so you don’t even write each MudTextField manually?

**ChatGPT said:**

Great question 👌 — this is where Blazor + reflection can save you from repeating a lot of <MudTextField> boilerplate.

You can dynamically enumerate the properties of your DTO, grab their [Display(Name="...")] (or property name if missing), and render a MudTextField for each.

**✅ Step 1. Your DTO with annotations**

using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required] [Display(Name = "Employee Name")] public string Name { get; set; } [Display(Name = "Department")] public string Department { get; set; } [Display(Name = "Email Address")] [EmailAddress] public string Email { get; set; } }

**✅ Step 2. Blazor Page with Reflection-based Field Rendering**

@page "/employee-dynamic" @using System.ComponentModel.DataAnnotations @using System.Reflection <EditForm Model="@employee" OnValidSubmit="OnValidSubmit"> <DataAnnotationsValidator /> <MudPaper Class="pa-4"> @foreach (var prop in typeof(EmployeeDto).GetProperties()) { var displayAttr = prop.GetCustomAttribute<DisplayAttribute>(); var label = displayAttr?.Name ?? prop.Name; <MudTextField @bind-Value="@GetValue(prop)" Label="@label" /> } <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Save </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private void OnValidSubmit() { // Save employee here } private string GetValue(PropertyInfo prop) { return prop.GetValue(employee)?.ToString(); } private void SetValue(PropertyInfo prop, object value) { prop.SetValue(employee, Convert.ChangeType(value, prop.PropertyType)); } }

**✅ Step 3. Fix Binding (**@bind-Value**)**

Since MudTextField requires **two-way binding**, you need a little trick:

<MudTextField Value="@GetValue(prop)" ValueChanged="@(val => SetValue(prop, val))" Label="@label" />

This way:

* Value is read via GetValue
* ValueChanged updates the employee object via SetValue

**🔑 What you get**

* Form auto-generates fields for every property in EmployeeDto
* Uses [Display(Name="...")] for labels
* Still works with [Required], [EmailAddress], etc. via <DataAnnotationsValidator />
* Minimal boilerplate → add properties to DTO, and UI updates automatically 🎉

⚠️ Note: This works best for **text fields**. If you want different field types (e.g., DateTime → MudDatePicker, bool → MudCheckBox, enum → MudSelect), you’d extend the loop with type checks.

👉 Do you want me to extend this so it auto-selects the **correct MudBlazor input control** based on property type (string → MudTextField, DateTime → MudDatePicker, bool → MudCheckBox, enum → MudSelect)?

**You said:**

Provide a sample data validation using an EditForm that contains 3 fields such as Employee Name, Date of Birth, and Email Address where Employee Name is required, Date of Birth must be in dd/MM/yyyy format, and Email Address should be valid email address and in correct format. The validation summary must be displayed at the top section of the razor page and will only be shown if there are validation error.

**ChatGPT said:**

Perfect 👍 — let’s build this step by step.

We’ll use:

* EditForm with DataAnnotationsValidator
* ValidationSummary at the top (only shown when errors exist)
* DataAnnotations for validation rules on the DTO

**✅ EmployeeDto.cs (Model with Validation)**

using System; using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required(ErrorMessage = "Employee Name is required")] [Display(Name = "Employee Name")] public string Name { get; set; } [Required(ErrorMessage = "Date of Birth is required")] [Display(Name = "Date of Birth")] [DataType(DataType.Date)] *// We'll validate format (dd/MM/yyyy) separately via a custom attribute* public string DateOfBirth { get; set; } [Required(ErrorMessage = "Email is required")] [EmailAddress(ErrorMessage = "Invalid Email Address")] [Display(Name = "Email Address")] public string Email { get; set; } }

**✅ Custom Date Format Validation Attribute**

using System; using System.ComponentModel.DataAnnotations; using System.Globalization; public class DateFormatAttribute : ValidationAttribute { private readonly string \_dateFormat; public DateFormatAttribute(string dateFormat) { \_dateFormat = dateFormat; ErrorMessage = $"Date must be in format {\_dateFormat}"; } protected override ValidationResult IsValid(object value, ValidationContext validationContext) { if (value is null) return ValidationResult.Success; string dateString = value.ToString(); if (DateTime.TryParseExact(dateString, \_dateFormat, CultureInfo.InvariantCulture, DateTimeStyles.None, out \_)) { return ValidationResult.Success; } return new ValidationResult(ErrorMessage); } }

Update the DTO to use it:

[Required(ErrorMessage = "Date of Birth is required")] [DateFormat("dd/MM/yyyy")] [Display(Name = "Date of Birth")] public string DateOfBirth { get; set; }

**✅ Employee.razor (Blazor Page)**

@page "/employee" @using System.ComponentModel.DataAnnotations <EditForm Model="@employee" OnValidSubmit="HandleValidSubmit"> <DataAnnotationsValidator /> @if (hasErrors) { <MudPaper Class="pa-2 mb-4"> <MudText Typo="Typo.subtitle2" Color="Color.Error">Validation Summary</MudText> <ValidationSummary /> </MudPaper> } <MudPaper Class="pa-4"> <MudTextField For="@(() => employee.Name)" /> <MudTextField For="@(() => employee.DateOfBirth)" Placeholder="dd/MM/yyyy" /> <MudTextField For="@(() => employee.Email)" /> <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Submit </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private bool hasErrors = false; private void HandleValidSubmit() { // If we got here, model is valid hasErrors = false; // Do save logic } protected override void OnParametersSet() { // Reset error display when navigating to page hasErrors = false; } // This will be triggered by validation errors automatically // Trick: ValidationSummary only shows if there are errors // So we set hasErrors = true if any validation fails protected override void OnAfterRender(bool firstRender) { if (firstRender) return; var context = new ValidationContext(employee); var results = new List<ValidationResult>(); hasErrors = !Validator.TryValidateObject(employee, context, results, true); } }

**🔑 Behavior**

* Employee Name → must not be empty.
* Date of Birth → must be in dd/MM/yyyy format (01/09/1990).
* Email Address → must be valid (e.g., user@example.com).
* If any validation fails → **Validation Summary** box shows at the top with all messages.
* If valid → form submits, hasErrors becomes false, summary is hidden.

👉 Would you like me to also make the **Date of Birth field use a**MudDatePicker (so users can pick a date, but still validate and display it in dd/MM/yyyy)?

**You said:**

make the Date of Birth field use a MudDatePicker (so users can pick a date, but still validate and display it in dd/MM/yyyy)

**ChatGPT said:**

Great choice 👍 using a MudDatePicker improves the UX while still keeping your strict validation.  
Here’s how we adapt your EmployeeDto, validation, and Employee.razor to work with MudDatePicker:

**✅ EmployeeDto.cs**

Change **DateOfBirth** type from string → DateTime? (nullable, so you can still validate if empty).

using System; using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required(ErrorMessage = "Employee Name is required")] [Display(Name = "Employee Name")] public string Name { get; set; } [Required(ErrorMessage = "Date of Birth is required")] [Display(Name = "Date of Birth")] [DataType(DataType.Date)] public DateTime? DateOfBirth { get; set; } [Required(ErrorMessage = "Email is required")] [EmailAddress(ErrorMessage = "Invalid Email Address")] [Display(Name = "Email Address")] public string Email { get; set; } }

**✅ Custom Date Format Validation (dd/MM/yyyy)**

Since MudDatePicker binds to a DateTime?, you don’t need users to manually type dd/MM/yyyy.  
But if you want to **ensure output is formatted as**dd/MM/yyyy, you can enforce it in the UI display.

**✅ Employee.razor**

@page "/employee" @using System.ComponentModel.DataAnnotations @using MudBlazor <EditForm Model="@employee" OnValidSubmit="HandleValidSubmit"> <DataAnnotationsValidator /> @if (hasErrors) { <MudPaper Class="pa-2 mb-4"> <MudText Typo="Typo.subtitle2" Color="Color.Error">Validation Summary</MudText> <ValidationSummary /> </MudPaper> } <MudPaper Class="pa-4"> <!-- Employee Name --> <MudTextField For="@(() => employee.Name)" /> <!-- Date of Birth with MudDatePicker --> <MudDatePicker @bind-Date="employee.DateOfBirth" Label="Date of Birth" DateFormat="dd/MM/yyyy" Clearable="true" Required="true" RequiredError="Date of Birth is required" /> <!-- Email --> <MudTextField For="@(() => employee.Email)" /> <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Submit </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private bool hasErrors = false; private void HandleValidSubmit() { hasErrors = false; // Save employee here } protected override void OnAfterRender(bool firstRender) { if (firstRender) return; var context = new ValidationContext(employee); var results = new List<ValidationResult>(); hasErrors = !Validator.TryValidateObject(employee, context, results, true); } }

**🔑 Behavior**

* **Employee Name** → required.
* **Date of Birth** → required, selected from date picker. Displays in dd/MM/yyyy.
* **Email Address** → must be valid.
* If errors → Validation Summary at the top.
* If valid → form submits successfully.

👉 Do you also want me to extend this so that the **ValidationSummary** auto-updates without needing OnAfterRender (using OnInvalidSubmit event instead)? That would simplify error handling.

**You said:**

Explain the following line of code in OnAfterRender method: if (firstRender) return;

**ChatGPT said:**

That’s a really good question 👍 Let’s break it down:

**🔹 The**OnAfterRender**method**

In Blazor, OnAfterRender(bool firstRender) is a **lifecycle method** that runs **after the component has rendered**.  
It has a parameter firstRender that tells you:

* true → this is the **very first time** the component rendered after it was initialized.
* false → this is a **subsequent render** (caused by state changes, events, or re-renders).

**🔹 The line of code:**

if (firstRender) return;

This means:

* **If this is the first render, exit the method immediately.**
* Skip executing the rest of the code until the next render.

**🔹 Why is this needed?**

1. **Avoid running logic too early**  
   Sometimes you only want code to run *after* the first render (e.g., validation checks, JS interop, DOM measurements).  
   Skipping the first render prevents errors like “element not found” or unnecessary API calls.
2. **Performance reasons**  
   Running expensive code (validation, database calls, caching) on first render may not make sense, since the model is still empty.  
   You usually want to run logic *after the user has interacted* and caused a re-render.
3. **Example in your case (validation summary)**
   * First render → form is blank, no errors. You don’t want to show validation yet.
   * Later render → user interacts (or submits), model changes → then you validate and possibly show the error summary.

So the check:

if (firstRender) return;

is a **guard clause** to say:  
➡️ “Skip validation on the first render, only validate after user interaction.”

✅ Without this check, you might see the Validation Summary flash with errors before the user even tries to submit.

👉 Do you want me to show you a version where you don’t even need OnAfterRender (instead using OnInvalidSubmit and OnValidSubmit), so validation logic is cleaner?

**You said:**

The validation summary does not appear if there is only 1 error in the data entry. It appears if 2 or more fields have error in the data entry.

**ChatGPT said:**

Ahh 👍 I see what’s happening. That behavior isn’t because of MudBlazor — it’s how Blazor’s built-in <ValidationSummary /> works.

**🔎 Why only shows with 2+ errors?**

* By default, <ValidationSummary /> shows **only model-level errors** (validation messages not tied to a specific property).
* When you have **just 1 property error**, Blazor puts that message on the field itself (via <ValidationMessage For="..." />) rather than in the summary.
* When you have **multiple errors**, Blazor also lists them at the top summary.

That’s why with one invalid field, your summary looks “empty.”

**✅ Fix #1 – Show all errors in the summary**

Tell the <ValidationSummary /> to also display **field-level errors**:

<ValidationSummary ShowFieldErrors="true" />

Now even if only one field is wrong (e.g., invalid email), you’ll see the error at the top.

**✅ Fix #2 – Combine field messages + summary**

If you want both inline error messages and a global summary, you can use:

<MudTextField For="@(() => employee.Name)" /> <ValidationMessage For="@(() => employee.Name)" /> <MudDatePicker @bind-Date="employee.DateOfBirth" Label="Date of Birth" DateFormat="dd/MM/yyyy" Required="true" /> <ValidationMessage For="@(() => employee.DateOfBirth)" /> <MudTextField For="@(() => employee.Email)" /> <ValidationMessage For="@(() => employee.Email)" />

And keep your <ValidationSummary /> at the top.  
This way:

* Users see specific errors under each field.
* The summary still shows a list at the top.

**✅ Fix #3 – Customizing Validation Display**

If you want **only** a top summary (no per-field messages), then use ShowFieldErrors="true" in the summary and **don’t include** <ValidationMessage> tags per field.

⚡ So in short:

* **1 error only → use**ShowFieldErrors="true"**.**
* **Multiple errors → works already, but**ShowFieldErrors="true"**makes it consistent.**

👉 Do you want me to rewrite your Employee.razor form with **just a top summary** (no per-field messages), so the behavior is consistent?

**You said:**

rewrite your Employee.razor form with just a top summary (no per-field messages), so the behavior is consistent?

**ChatGPT said:**

Perfect 👍 If you want your **validation errors to only appear in the top summary** (and not inline for each field), then we just need to:

1. Use <ValidationSummary ShowFieldErrors="true" /> at the top.
2. Remove all <ValidationMessage For="..."> entries.
3. Keep For="..." in your MudTextField / MudDatePicker so Blazor knows which property to validate.

**✅ Employee.razor (clean version with top-only summary)**

@page "/employee" @using System.ComponentModel.DataAnnotations @using MudBlazor <EditForm Model="@employee" OnValidSubmit="HandleValidSubmit" OnInvalidSubmit="HandleInvalidSubmit"> <DataAnnotationsValidator /> @if (showValidationSummary) { <MudPaper Class="pa-2 mb-4"> <MudText Typo="Typo.subtitle2" Color="Color.Error">Validation Summary</MudText> <ValidationSummary ShowFieldErrors="true" /> </MudPaper> } <MudPaper Class="pa-4"> <!-- Employee Name --> <MudTextField For="@(() => employee.Name)" /> <!-- Date of Birth --> <MudDatePicker For="@(() => employee.DateOfBirth)" DateFormat="dd/MM/yyyy" Clearable="true" /> <!-- Email --> <MudTextField For="@(() => employee.Email)" /> <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Submit </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private bool showValidationSummary = false; private void HandleValidSubmit() { showValidationSummary = false; // hide summary if valid // save logic here } private void HandleInvalidSubmit(EditContext context) { showValidationSummary = true; // show summary only when invalid } }

**✅ EmployeeDto.cs**

using System; using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required(ErrorMessage = "Employee Name is required")] [Display(Name = "Employee Name")] public string Name { get; set; } [Required(ErrorMessage = "Date of Birth is required")] [Display(Name = "Date of Birth")] [DataType(DataType.Date)] public DateTime? DateOfBirth { get; set; } [Required(ErrorMessage = "Email is required")] [EmailAddress(ErrorMessage = "Invalid Email Address")] [Display(Name = "Email Address")] public string Email { get; set; } }

**🔑 Behavior**

* All validation messages (even for a single field) appear **only in the top summary**.
* No inline field-level messages are shown.
* Summary only appears when there are errors.
* Summary hides automatically on successful submit.

👉 Do you also want me to show how to style the summary (e.g., list with red bullets, or a MudAlert instead of plain MudText)?

**You said:**

Setting <ValidationSummary ShowFieldErrors="true" /> still does not display validation summary for 1 property error

**ChatGPT said:**

Got it 👍 Thanks for pointing this out — this is a known quirk in **Blazor’s built-in**ValidationSummary (not MudBlazor). Even with ShowFieldErrors="true", sometimes property-level errors don’t surface in the summary until there are multiple.

👉 The cleanest workaround is to **build your own Validation Summary** that explicitly lists **all property errors from the**EditContext. That way, you have full control.

**✅ Custom Validation Summary Component**

First, create a reusable component:

**Shared/CustomValidationSummary.razor**

@using Microsoft.AspNetCore.Components.Forms @inject IServiceProvider Services @if (Messages?.Any() == true) { <MudPaper Class="pa-2 mb-4"> <MudText Typo="Typo.subtitle2" Color="Color.Error">Validation Summary</MudText> <ul class="mud-list pl-4"> @foreach (var message in Messages) { <li>@message</li> } </ul> </MudPaper> } @code { [CascadingParameter] EditContext CurrentEditContext { get; set; } private List<string> Messages = new(); protected override void OnInitialized() { if (CurrentEditContext == null) { throw new InvalidOperationException($"{nameof(CustomValidationSummary)} requires a cascading " + $"parameter of type {nameof(EditContext)}."); } CurrentEditContext.OnValidationStateChanged += (s, e) => { Messages = CurrentEditContext .GetValidationMessages() .ToList(); StateHasChanged(); }; } }

**✅ Employee.razor**

Now replace the default <ValidationSummary> with your custom one:

@page "/employee" @using System.ComponentModel.DataAnnotations @using MudBlazor <EditForm Model="@employee" OnValidSubmit="HandleValidSubmit" OnInvalidSubmit="HandleInvalidSubmit"> <DataAnnotationsValidator /> @if (showValidationSummary) { <CustomValidationSummary /> } <MudPaper Class="pa-4"> <MudTextField For="@(() => employee.Name)" /> <MudDatePicker For="@(() => employee.DateOfBirth)" DateFormat="dd/MM/yyyy" Clearable="true" /> <MudTextField For="@(() => employee.Email)" /> <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Submit </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private bool showValidationSummary = false; private void HandleValidSubmit() { showValidationSummary = false; // Save logic } private void HandleInvalidSubmit(EditContext context) { showValidationSummary = true; } }

**🔑 What changed?**

* The default Blazor <ValidationSummary> is replaced by **CustomValidationSummary**.
* CustomValidationSummary subscribes to OnValidationStateChanged and pulls *all* error messages (including single property errors).
* Now, even **1 invalid field** shows up in the summary consistently.

⚡ This approach also lets you fully style the summary (e.g., use MudAlert or MudList) instead of the plain <ul>.

👉 Do you want me to adapt this so the summary shows inside a **MudAlert (red error box)** instead of a bullet list? That would make it look cleaner in MudBlazor UI.

**You said:**

Tell me how to change the font color of the ValidationSummary error messages

**ChatGPT said:**

Good question 👍 By default, <ValidationSummary /> just dumps plain <div> elements with error messages — no styling. To change font color, you have a couple of options depending on whether you’re using the built-in ValidationSummary or the custom one we built earlier.

**✅ Option 1 – Style Built-in**<ValidationSummary />

You can wrap it in a MudText or a div with a CSS class:

<MudPaper Class="pa-2 mb-4"> <MudText Typo="Typo.subtitle2" Color="Color.Error">Validation Summary</MudText> <div class="validation-errors"> <ValidationSummary ShowFieldErrors="true" /> </div> </MudPaper>

Then in your CSS (e.g., wwwroot/css/site.css or scoped CSS):

.validation-errors { color: red; */\* or use a MudBlazor theme color \*/* }

**✅ Option 2 – Use**MudText**Inside a Custom Summary (Recommended)**

If you use the **CustomValidationSummary** from before, you can render each message as a MudText and set the color directly:

@if (Messages?.Any() == true) { <MudPaper Class="pa-2 mb-4"> <MudText Typo="Typo.subtitle2" Color="Color.Error">Validation Summary</MudText> <ul class="mud-list pl-4"> @foreach (var message in Messages) { <li> <MudText Color="Color.Error">@message</MudText> </li> } </ul> </MudPaper> }

This way, each error message will automatically get the **MudBlazor error red** color.

**✅ Option 3 – Use**MudAlert**(Nice UX)**

Another clean way is to wrap all errors in a MudAlert:

@if (Messages?.Any() == true) { <MudAlert Severity="Severity.Error" Variant="Variant.Outlined"> <ul class="pl-4"> @foreach (var message in Messages) { <li>@message</li> } </ul> </MudAlert> }

Here, the font color and background automatically match MudBlazor’s error theme.

🔑 **Summary**:

* Use **CSS** if you stick with Blazor’s built-in ValidationSummary.
* Use **MudText / MudAlert** if you’re already customizing error rendering in MudBlazor — more consistent and theme-aware.

👉 Do you want me to rewrite your current CustomValidationSummary so that it uses **MudAlert with red text** automatically (no CSS needed)?

**You said:**

I have a property in my Employee class called OfficialEmail. I set the [EmailAddress] data annotation attribute. However, when I run the application, it throws data validation error that says: The Official Email field is not a valid e-mail address, though this property is not mandatory where there is no [Required] attribute set.

**ChatGPT said:**

That’s a subtle but very common gotcha 👍 Let’s unpack it carefully.

**🔎 Why it happens**

When you decorate a property with [EmailAddress], Blazor’s DataAnnotationsValidator will run **whenever that property has a value**.

* If the property is null or empty → it **should pass** because [EmailAddress] does not imply required.
* If the property has text → it must match an email regex pattern.
* If you leave the field empty in a MudTextField, depending on binding, the component might not give you null — it might give you "" (empty string).
  + Empty string "" is still “a value,” so the [EmailAddress] validator tries to validate it.
  + "" is not a valid email → validation fails.

That’s why you see the error even though [Required] is not set.

**✅ Fixes**

You have a few clean options:

**Option 1 – Allow empty string explicitly**

Add [EmailAddress] **and** [Required(AllowEmptyStrings = true)]:

[EmailAddress(ErrorMessage = "The Official Email field is not a valid e-mail address")] [Required(AllowEmptyStrings = true)] public string OfficialEmail { get; set; }

This tells validation:

* Empty string is fine.
* Non-empty must be a valid email.

**Option 2 – Change to**string?**and set**OfficialEmail = null**when blank**

If you don’t want empty string "", you can normalize it to null in your model binder or input. For example:

<MudTextField For="@(() => employee.OfficialEmail)" @bind-Value="employee.OfficialEmail" Immediate="true" OnBlur="@(e => NormalizeOfficialEmail())" />

And in code:

private void NormalizeOfficialEmail() { if (string.IsNullOrWhiteSpace(employee.OfficialEmail)) employee.OfficialEmail = null; }

Now [EmailAddress] only runs if not null.

**Option 3 – Custom validation attribute**

If you want full control, create a custom attribute:

public class OptionalEmailAttribute : ValidationAttribute { protected override ValidationResult IsValid(object value, ValidationContext validationContext) { var str = value as string; if (string.IsNullOrWhiteSpace(str)) return ValidationResult.Success; *// allow empty* var emailAttr = new EmailAddressAttribute(); return emailAttr.IsValid(str) ? ValidationResult.Success : new ValidationResult($"{validationContext.DisplayName} is not a valid email address"); } }

Usage:

[OptionalEmail] public string OfficialEmail { get; set; }

**🔑 Recommended approach**

If this is just a **standard optional email field**, use **Option 1** (simplest, built-in):

[EmailAddress] [Required(AllowEmptyStrings = true)] public string OfficialEmail { get; set; }

👉 Do you want me to show how to make your **Employee.razor form respect this optional email rule** so that:

* No error if left blank,
* Error only if a non-email value is typed?

**You said:**

show how to make your Employee.razor form respect this optional email rule so that: No error if left blank, Error only if a non-email value is typed?

**ChatGPT said:**

Perfect 👍 Let’s implement this cleanly in your **Employee.razor form** so that:

* ✅ OfficialEmail can be empty (no [Required])
* ✅ If the user types something, it must be a valid email

**✅ Step 1 – Update EmployeeDto**

Use [EmailAddress] **together with**[Required(AllowEmptyStrings = true)].  
This combination ensures *empty is allowed, but non-empty must validate*.

using System; using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required(ErrorMessage = "Employee Name is required")] [Display(Name = "Employee Name")] public string Name { get; set; } [Required(ErrorMessage = "Date of Birth is required")] [Display(Name = "Date of Birth")] [DataType(DataType.Date)] public DateTime? DateOfBirth { get; set; } [EmailAddress(ErrorMessage = "The Official Email field is not a valid e-mail address")] [Required(AllowEmptyStrings = true)] [Display(Name = "Official Email")] public string OfficialEmail { get; set; } }

**✅ Step 2 – Employee.razor**

Now add the field to your EditForm.  
We’ll keep only **top-level summary** (consistent with your earlier requirement):

@page "/employee" @using System.ComponentModel.DataAnnotations @using MudBlazor <EditForm Model="@employee" OnValidSubmit="HandleValidSubmit" OnInvalidSubmit="HandleInvalidSubmit"> <DataAnnotationsValidator /> @if (showValidationSummary) { <MudAlert Severity="Severity.Error" Variant="Variant.Outlined" Class="mb-4"> <ValidationSummary ShowFieldErrors="true" /> </MudAlert> } <MudPaper Class="pa-4"> <!-- Employee Name --> <MudTextField For="@(() => employee.Name)" /> <!-- Date of Birth --> <MudDatePicker For="@(() => employee.DateOfBirth)" DateFormat="dd/MM/yyyy" Clearable="true" /> <!-- Official Email (optional but validated if filled) --> <MudTextField For="@(() => employee.OfficialEmail)" /> <MudButton Type="Submit" Color="Color.Primary" Variant="Variant.Filled"> Submit </MudButton> </MudPaper> </EditForm> @code { private EmployeeDto employee = new(); private bool showValidationSummary = false; private void HandleValidSubmit() { showValidationSummary = false; // Save logic here } private void HandleInvalidSubmit(EditContext context) { showValidationSummary = true; } }

**🔑 Behavior**

* **Leave**OfficialEmail**blank** → ✅ no error.
* **Type invalid email (e.g., "abc")** → ❌ error: *“The Official Email field is not a valid e-mail address”*.
* **Type valid email (e.g., "abc@domain.com")** → ✅ accepted.

👉 Do you want me to also show you how to make OfficialEmail **nullable (string?)** so it saves null instead of "" when left empty? That can be useful if you want the database column to remain NULL instead of storing an empty string.

**You said:**

show me how to make OfficialEmail nullable (string?) so it saves null instead of "" when left empty? That can be useful if you want the database column to remain NULL instead of storing an empty string.

**ChatGPT said:**

Exactly 👍 This is a really good point — in EF Core (and most DBs), it’s usually cleaner to store NULL**instead of**"" for optional fields like email.

By default, Blazor’s MudTextField binds to a string and will send you "" when the input is empty. To fix this, you need to:

1. Make the property nullable (string?).
2. Normalize empty string ("") to null when binding.

**✅ Step 1 – Update EmployeeDto**

using System; using System.ComponentModel.DataAnnotations; public class EmployeeDto { [Required(ErrorMessage = "Employee Name is required")] [Display(Name = "Employee Name")] public string Name { get; set; } [Required(ErrorMessage = "Date of Birth is required")] [Display(Name = "Date of Birth")] [DataType(DataType.Date)] public DateTime? DateOfBirth { get; set; } [EmailAddress(ErrorMessage = "The Official Email field is not a valid e-mail address")] [Required(AllowEmptyStrings = true)] [Display(Name = "Official Email")] public string? OfficialEmail { get; set; } *// 👈 nullable string* }

**✅ Step 2 – Normalize Binding in Employee.razor**

MudTextField has a ValueChanged callback.  
We can intercept it and convert "" → null.

<MudTextField T="string" Label="Official Email" Value="@employee.OfficialEmail" ValueChanged="@(e => employee.OfficialEmail = string.IsNullOrWhiteSpace(e) ? null : e)" For="@(() => employee.OfficialEmail)" />

**🔑 What happens now**

* User leaves the field blank → employee.OfficialEmail == null → EF will save NULL.
* User types spaces → trimmed to null as well.
* User enters valid email → saved as string.
* Validation still works:
  + Empty/null → ✅ allowed.
  + Invalid string (e.g., "test") → ❌ validation error.

**✅ Step 3 – EF Core Migration (if needed)**

Make sure your DB column is nullable:

builder.Property(e => e.OfficialEmail) .HasMaxLength(255) .IsRequired(false); *// ✅ optional (nullable)*

✨ Now your OfficialEmail behaves exactly as an **optional email field** should:

* Stores NULL when empty,
* Validates only if filled,
* Saves valid emails correctly.

👉 Do you also want me to show how to apply this **empty-to-null normalization** for **all optional string fields automatically** (via a custom InputText/MudTextField wrapper), so you don’t repeat the ValueChanged everywhere?

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info.