# ASP.NET Core Blazor data binding

Article • 05/06/2024

This article explains data binding features for Razor components and DOM elements in Blazor apps.

## Binding features

Razor components provide data binding features with the @bind Razor directive attribute with a field, property, or Razor expression value.

The following example binds:

- An `<input>` element value to the C# `inputValue` field.
- A second `<input>` element value to the C# `InputValue` property.

When an `<input>` element loses focus, its bound field or property is updated.

`Bind.razor`:

razor

```razor
@page "/bind"

<PageTitle>Bind</PageTitle>

<h1>Bind Example</h1>

<p>
    <label>
        inputValue:
        <input @bind="inputValue" />
    </label>
</p>

<p>
    <label>
        InputValue:
        <input @bind="InputValue" />
    </label>
</p>

<ul>
    <li><code>inputValue</code>: @inputValue</li>
    <li><code>InputValue</code>: @InputValue</li>
</ul>

@code {
    private string? inputValue;

    private string? InputValue { get; set; }
}
```

The text box is updated in the UI only when the component is rendered, not in response to changing the field's or property's value. Since components render themselves after event handler code executes, field and property updates are usually reflected in the UI immediately after an event handler is triggered.

As a demonstration of how data binding composes in HTML, the following example binds the `InputValue` property to the second `<input>` element's `value` and `onchange` attributes (change). *The second `<input>` element in the following example is a concept demonstration and isn't meant to suggest how you should bind data in Razor components.*

`BindTheory.razor`:

razor

```razor
@page "/bind-theory"

<PageTitle>Bind Theory</PageTitle>

<h1>Bind Theory Example</h1>

<p>
    <label>
        Normal Blazor binding:
        <input @bind="InputValue" />
    </label>
</p>

<p>
    <label>
        Demonstration of equivalent HTML binding:
        <input value="@InputValue"
            @onchange="@((ChangeEventArgs __e) => InputValue = __e?.Value?.ToString())" />
    </label>
</p>

<p>
    <code>InputValue</code>: @InputValue
</p>

@code {
    private string? InputValue { get; set; }
}
```

When the `BindTheory` component is rendered, the `value` of the HTML demonstration `<input>` element comes from the `InputValue` property. When the user enters a value in the text box and changes element focus, the `onchange` event is fired and the `InputValue` property is set to the changed value. In reality, code execution is more complex because @bind handles cases where type conversions are performed. In general, @bind associates the current value of an expression with the `value` attribute of the `<input>` and handles changes using the registered handler.

Bind a property or field on other DOM events by including an `@bind:event="{EVENT}"` attribute with a DOM event for the `{EVENT}` placeholder. The following example binds the `InputValue` property to the `<input>` element's value when the element's `oninput` event (input) is triggered. Unlike the `onchange` event (change), which fires when the element loses focus, `oninput` (input) fires when the value of the text box changes.

`Page/BindEvent.razor`:

razor

```razor
@page "/bind-event"

<PageTitle>Bind Event</PageTitle>

<h1>Bind Event Example</h1>
```

```
<p>
    <label>
        InputValue:
        <input @bind="InputValue" @bind:event="oninput" />
    </label>

</p>

<p>
    <code>InputValue</code>: @InputValue
</p>

@code {
    private string? InputValue { get; set; }
}
```

To execute asynchronous logic after binding, use `@bind:after="{EVENT}"` with a DOM event for the `{EVENT}` placeholder. An assigned C# method isn't executed until the bound value is assigned synchronously.

Using an event callback parameter (EventCallback/EventCallback<T>) with `@bind:after` isn't supported. Instead, pass a method that returns an Action or Task to `@bind:after`.

In the following example:

- Each `<input>` element's `value` is bound to the `searchText` field synchronously.
- The `PerformSearch` method executes asynchronously:
  - When the first box loses focus (`onchange` event) after the value is changed.
  - After each keystroke (`oninput` event) in the second box.
- `PerformSearch` calls a service with an asynchronous method (`FetchAsync`) to return search results.

razor

```
@inject ISearchService SearchService

<input @bind="searchText" @bind:after="PerformSearch" />
<input @bind="searchText" @bind:event="oninput" @bind:after="PerformSearch" />

@code {
    private string? searchText;
    private string[]? searchResult;

    private async Task PerformSearch()
    {
        searchResult = await SearchService.FetchAsync(searchText);
    }
}
```

Additional examples

`BindAfter.razor`:

razor

```
@page "/bind-after"
@using Microsoft.AspNetCore.Components.Forms

<h1>Bind After Examples</h1>
```

```
<h2>Elements</h2>

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind="text" @bind:after="After" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<h2>Components</h2>

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value="text" @bind-Value:after="After" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

@code {
    private string text = "";

    private void After() {}
    private Task AfterAsync() { return Task.CompletedTask; }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

Components support two-way data binding by defining a pair of parameters:

- `@bind:get`: Specifies the value to bind.
- `@bind:set`: Specifies a callback for when the value changes.

The `@bind:get` and `@bind:set` modifiers are always used together.

Examples

`BindGetSet.razor`:

razor

```
@page "/bind-get-set"
@using Microsoft.AspNetCore.Components.Forms

<h1>Bind Get Set Examples</h1>

<h2>Elements</h2>

<input type="text" @bind:get="text" @bind:set="(value) => { text = value; }" />
<input type="text" @bind:get="text" @bind:set="Set" />
<input type="text" @bind:get="text" @bind:set="SetAsync" />

<h2>Components</h2>

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { text = value; }" />
<InputText @bind-Value:get="text" @bind-Value:set="Set" />
<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

@code {
    private string text = "";

    private void Set(string value)
    {
        text = value;
    }
```

```
    private Task SetAsync(string value)
    {
        text = value;
        return Task.CompletedTask;
    }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

For another example use of `@bind:get` and `@bind:set`, see the Bind across more than two components section later in this article.

Razor attribute binding is case-sensitive:

- `@bind`, `@bind:event`, and `@bind:after` are valid.
- `@Bind`/`@bind:Event`/`@bind:aftEr` (capital letters) or `@BIND`/`@BIND:EVENT`/`@BIND:AFTER` (all capital letters) **are invalid**.

## Use `@bind:get`/`@bind:set` modifiers and avoid event handlers for two-way data binding

Two-way data binding isn't possible to implement with an event handler. Use `@bind:get`/`@bind:set` modifiers for two-way data binding.

❌ Consider the following ***dysfunctional approach*** for two-way data binding using an event handler:

razor

```
<p>
    <input value="@inputValue" @oninput="OnInput" />
</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private string? inputValue;

    private void OnInput(ChangeEventArgs args)
    {
        var newValue = args.Value?.ToString() ?? string.Empty;

        inputValue = newValue.Length > 4 ? "Long!" : newValue;
    }
}
```

The `OnInput` event handler updates the value of `inputValue` to `Long!` after a fourth character is provided. However, the user can continue adding characters to the element value in the UI. The value of `inputValue` isn't bound back to the element's value with each keystroke. The preceding example is only capable of one-way data binding.

The reason for this behavior is that Blazor isn't aware that your code intends to modify the value of `inputValue` in the event handler. Blazor doesn't try to force DOM element values and .NET variable values to match unless they're bound with `@bind` syntax. In earlier versions of Blazor, two-way data binding is implemented by binding the element to a

property and controlling the property's value with its setter. In ASP.NET Core in .NET 7 or later, `@bind:get`/`@bind:set` modifier syntax is used to implement two-way data binding, as the next example demonstrates.

✔ Consider the following ***correct approach*** using `@bind:get`/`@bind:set` for two-way data binding:

razor

```razor
<p>
    <input @bind:event="oninput" @bind:get="inputValue" @bind:set="OnInput" />
</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private string? inputValue;

    private void OnInput(string value)
    {
        var newValue = value ?? string.Empty;

        inputValue = newValue.Length > 4 ? "Long!" : newValue;
    }
}
```

Using `@bind:get`/`@bind:set` modifiers both controls the underlying value of `inputValue` via `@bind:set` and binds the value of `inputValue` to the element's value via `@bind:get`. The preceding example demonstrates the correct approach for implementing two-way data binding.

## Binding to a property with C# `get` and `set` accessors

C# get and set accessors can be used to create custom binding format behavior, as the following `DecimalBinding` component demonstrates. The component binds a positive or negative decimal with up to three decimal places to an `<input>` element by way of a `string` property (`DecimalValue`).

`DecimalBinding.razor`:

razor

```razor
@page "/decimal-binding"
@using System.Globalization

<PageTitle>Decimal Binding</PageTitle>

<h1>Decimal Binding Example</h1>

<p>
    <label>
        Decimal value (±0.000 format):
        <input @bind="DecimalValue" />
    </label>
</p>

<p>
    <code>decimalValue</code>: @decimalValue
</p>
```

```
@code {
    private decimal decimalValue = 1.1M;
    private NumberStyles style =
        NumberStyles.AllowDecimalPoint | NumberStyles.AllowLeadingSign;
    private CultureInfo culture = CultureInfo.CreateSpecificCulture("en-US");

    private string DecimalValue
    {
        get => decimalValue.ToString("0.000", culture);
        set
        {
            if (Decimal.TryParse(value, style, culture, out var number))
            {
                decimalValue = Math.Round(number, 3);
            }
        }
    }
}
```

> **Note**
>
> Two-way binding to a property with `get`/`set` accessors requires discarding the Task returned by EventCallback.InvokeAsync. For two-way data binding, we recommend using `@bind:get`/`@bind:set` modifiers. For more information, see the `@bind:get`/`@bind:set` guidance in the earlier in this article.

# Multiple option selection with `<select>` elements

Binding supports multiple option selection with `<select>` elements. The @onchange event provides an array of the selected elements via event arguments (ChangeEventArgs). The value must be bound to an array type.

`BindMultipleInput.razor`:

razor

```
@page "/bind-multiple-input"

<h1>Bind Multiple <code>input</code>Example</h1>

<p>
    <label>
        Select one or more cars:
        <select @onchange="SelectedCarsChanged" multiple>
            <option value="audi">Audi</option>
            <option value="jeep">Jeep</option>
            <option value="opel">Opel</option>
            <option value="saab">Saab</option>
            <option value="volvo">Volvo</option>
        </select>
    </label>
</p>

<p>
    Selected Cars: @string.Join(", ", SelectedCars)
</p>

<p>
    <label>
```

```
        Select one or more cities:
        <select @bind="SelectedCities" multiple>
            <option value="bal">Baltimore</option>
            <option value="la">Los Angeles</option>
            <option value="pdx">Portland</option>
            <option value="sf">San Francisco</option>
            <option value="sea">Seattle</option>
        </select>
    </label>
</p>

<span>
    Selected Cities: @string.Join(", ", SelectedCities)
</span>

@code {
    public string[] SelectedCars { get; set; } = new string[] { };
    public string[] SelectedCities { get; set; } = new[] { "bal", "sea" };

    private void SelectedCarsChanged(ChangeEventArgs e)
    {
        if (e.Value is not null)
        {
            SelectedCars = (string[])e.Value;
        }
    }
}
```

For information on how empty strings and `null` values are handled in data binding, see the Binding <select> element options to C# object null values section.

# Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

# Unparsable values

When a user provides an unparsable value to a data-bound element, the unparsable value is automatically reverted to its previous value when the bind event is triggered.

Consider the following component, where an `<input>` element is bound to an `int` type with an initial value of `123`.

`UnparsableValues.razor`:

razor

```razor
@page "/unparsable-values"

<PageTitle>Unparsable Values</PageTitle>

<h1>Unparsable Values Example</h1>

<p>
    <label>
        inputValue:
        <input @bind="inputValue" />
    </label>

</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private int inputValue = 123;
}
```

By default, binding applies to the element's `onchange` event. If the user updates the value of the text box's entry to `123.45` and changes the focus, the element's value is reverted to `123` when `onchange` fires. When the value `123.45` is rejected in favor of the original value of `123`, the user understands that their value wasn't accepted.

For the `oninput` event (`@bind:event="oninput"`), a value reversion occurs after any keystroke that introduces an unparsable value. When targeting the `oninput` event with an `int`-bound type, a user is prevented from typing a dot (`.`) character. A dot (`.`) character is immediately removed, so the user receives immediate feedback that only whole numbers are permitted. There are scenarios where reverting the value on the `oninput` event isn't ideal, such as when the user should be allowed to clear an unparsable `<input>` value. Alternatives include:

- Don't use the `oninput` event. Use the default `onchange` event, where an invalid value isn't reverted until the element loses focus.
- Bind to a nullable type, such as `int?` or `string` and either use `@bind:get`/`@bind:set` modifiers (described earlier in this article) or bind to a property with custom get and set accessor logic to handle invalid entries.
- Use an input component, such as InputNumber<TValue> or InputDate<TValue>, with form validation. Input components together with form validation components provide built-in support to manage invalid inputs:
  - Permit the user to provide invalid input and receive validation errors on the associated EditContext.
  - Display validation errors in the UI without interfering with the user entering additional webform data.

# Format strings

Data binding works with a single DateTime format string using `@bind:format="{FORMAT STRING}"`, where the `{FORMAT STRING}` placeholder is the format string. Other format expressions, such as currency or number formats, aren't available at this time but might be added in a future release.

`DateBinding.razor`:

```razor
@page "/date-binding"

<PageTitle>Date Binding</PageTitle>

<h1>Date Binding Example</h1>

<p>
    <label>
        <code>yyyy-MM-dd</code> format:
        <input @bind="startDate" @bind:format="yyyy-MM-dd" />
    </label>
</p>

<p>
    <code>startDate</code>: @startDate
</p>

@code {
    private DateTime startDate = new(2020, 1, 1);
}
```

In the preceding code, the `<input>` element's field type (`type` attribute) defaults to `text`.

Nullable System.DateTime and System.DateTimeOffset are supported:

```C#
private DateTime? date;
private DateTimeOffset? dateOffset;
```

Specifying a format for the `date` field type isn't recommended because Blazor has built-in support to format dates. In spite of the recommendation, only use the `yyyy-MM-dd` date format for binding to function correctly if a format is supplied with the `date` field type:

```razor
<input type="date" @bind="startDate" @bind:format="yyyy-MM-dd">
```

# Binding with component parameters

A common scenario is binding a property of a child component to a property in its parent component. This scenario is called a *chained bind* because multiple levels of binding occur simultaneously.

You can't implement chained binds with @bind syntax in a child component. An event handler and value must be specified separately to support updating the property in the parent from the child component. The parent component still leverages @bind syntax to set up data binding with the child component.

The following `ChildBind` component has a `Year` component parameter and an EventCallback<TValue>. By convention, the EventCallback<TValue> for the parameter must be named as the component parameter name with a "`Changed`" suffix. The

naming syntax is `{PARAMETER NAME}Changed`, where the `{PARAMETER NAME}` placeholder is the parameter name. In the following example, the EventCallback<TValue> is named `YearChanged`.

EventCallback.InvokeAsync invokes the delegate associated with the binding with the provided argument and dispatches an event notification for the changed property.

`ChildBind.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:18rem ">
    <div class="card-body">
        <h3 class="card-title">ChildBind Component</h3>
        <p class="card-text">
            Child <code>Year</code>: @Year
        </p>
        <button @onclick="UpdateYearFromChild">Update Year from Child</button>
    </div>
</div>

@code {
    [Parameter]
    public int Year { get; set; }

    [Parameter]
    public EventCallback<int> YearChanged { get; set; }

    private async Task UpdateYearFromChild()
    {
        await YearChanged.InvokeAsync(Random.Shared.Next(1950, 2021));
    }
}
```

For more information on events and EventCallback<TValue>, see the *EventCallback* section of the ASP.NET Core Blazor event handling article.

In the following `Parent1` component, the `year` field is bound to the `Year` parameter of the child component. The `Year` parameter is bindable because it has a companion `YearChanged` event that matches the type of the `Year` parameter.

`Parent1.razor`:

razor

```razor
@page "/parent-1"

<PageTitle>Parent 1</PageTitle>

<h1>Parent Example 1</h1>

<p>Parent <code>year</code>: @year</p>

<button @onclick="UpdateYear">Update Parent <code>year</code></button>

<ChildBind @bind-Year="year" />

@code {
    private int year = 1979;

    private void UpdateYear()
    {
```

```
        year = Random.Shared.Next(1950, 2021);
    }
}
```

Component parameter binding can also trigger `@bind:after` events. In the following example, the `YearUpdated` method executes asynchronously after binding the `Year` component parameter.

razor

```razor
<ChildBind @bind-Year="year" @bind-Year:after="YearUpdated" />

@code {
    ...

    private async Task YearUpdated()
    {
        ... = await ...;
    }
}
```

By convention, a property can be bound to a corresponding event handler by including an `@bind-{PROPERTY}:event` attribute assigned to the handler, where the `{PROPERTY}` placeholder is the property. `<ChildBind @bind-Year="year" />` is equivalent to writing:

razor

```razor
<ChildBind @bind-Year="year" @bind-Year:event="YearChanged" />
```

In a more sophisticated and real-world example, the following `PasswordEntry` component:

- Sets an `<input>` element's value to a `password` field.
- Exposes changes of a `Password` property to a parent component with an EventCallback that passes in the current value of the child's `password` field as its argument.
- Uses the `onclick` event to trigger the `ToggleShowPassword` method. For more information, see ASP.NET Core Blazor event handling.

`PasswordEntry.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:22rem ">
    <div class="card-body">
        <h3 class="card-title">Password Component</h3>
        <p class="card-text">
            <label>
                Password:
                <input @oninput="OnPasswordChanged"
                       required
                       type="@(showPassword ? "text" : "password")"
                       value="@password" />
            </label>
        </p>
        <button class="btn btn-primary" @onclick="ToggleShowPassword">
            Show password
        </button>
    </div>
```

```
</div>

@code {
    private bool showPassword;
    private string? password;

    [Parameter]
    public string? Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private async Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e?.Value?.ToString();

        await PasswordChanged.InvokeAsync(password);
    }

    private void ToggleShowPassword()
    {
        showPassword = !showPassword;
    }
}
```

The `PasswordEntry` component is used in another component, such as the following `PasswordBinding` component example.

`PasswordBinding.razor`:

razor

```
@page "/password-binding"

<PageTitle>Password Binding</PageTitle>

<h1>Password Binding Example</h1>

<PasswordEntry @bind-Password="password" />

<p>
    <code>password</code>: @password
</p>

@code {
    private string password = "Not set";
}
```

When the `PasswordBinding` component is initially rendered, the `password` value of `Not set` is displayed in the UI. After initial rendering, the value of `password` reflects changes made to the `Password` component parameter value in the `PasswordEntry` component.

> **Note**
>
> The preceding example binds the password one-way from the child `PasswordEntry` component to the parent `PasswordBinding` component. Two-way binding isn't a requirement in this scenario if the goal is for the app to have a shared password entry component for reuse around the app that merely passes the password to the parent. For an

approach that permits two-way binding without writing directly to the child component's parameter, see the `NestedChild` component example in the Bind across more than two components section of this article.

Perform checks or trap errors in the handler. The following revised `PasswordEntry` component provides immediate feedback to the user if a space is used in the password's value.

`PasswordEntry.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:22rem ">
    <div class="card-body">
        <h3 class="card-title">Password Component</h3>
        <p class="card-text">
            <label>
                Password:
                <input @oninput="OnPasswordChanged"
                       required
                       type="@(showPassword ? "text" : "password")"
                       value="@password" />
            </label>
            <span class="text-danger">@validationMessage</span>
        </p>
        <button class="btn btn-primary" @onclick="ToggleShowPassword">
            Show password
        </button>
    </div>
</div>

@code {
    private bool showPassword;
    private string? password;
    private string? validationMessage;

    [Parameter]
    public string? Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e?.Value?.ToString();

        if (password != null && password.Contains(' '))
        {
            validationMessage = "Spaces not allowed!";

            return Task.CompletedTask;
        }
        else
        {
            validationMessage = string.Empty;

            return PasswordChanged.InvokeAsync(password);
        }
    }

    private void ToggleShowPassword()
    {
        showPassword = !showPassword;
```

```
    }
}
```

# Bind across more than two components

You can bind parameters through any number of nested components, but you must respect the one-way flow of data:

- Change notifications *flow up the hierarchy*.
- New parameter values *flow down the hierarchy*.

A common and recommended approach is to only store the underlying data in the parent component to avoid any confusion about what state must be updated, as shown in the following example.

`Parent2.razor`:

razor

```razor
@page "/parent-2"

<PageTitle>Parent 2</PageTitle>

<h1>Parent Example 2</h1>

<p>Parent Message: <b>@parentMessage</b></p>

<p>
    <button @onclick="ChangeValue">Change from Parent</button>
</p>

<NestedChild @bind-ChildMessage="parentMessage" />

@code {
    private string parentMessage = "Initial value set in Parent";

    private void ChangeValue()
    {
        parentMessage = $"Set in Parent {DateTime.Now}";
    }
}
```

In the following `NestedChild` component, the `NestedGrandchild` component:

- Assigns the value of `ChildMessage` to `GrandchildMessage` with `@bind:get` syntax.
- Updates `GrandchildMessage` when `ChildMessageChanged` executes with `@bind:set` syntax.

`NestedChild.razor`:

razor

```razor
<div class="border rounded m-1 p-1">
    <h2>Child Component</h2>

    <p>Child Message: <b>@ChildMessage</b></p>

    <p>
        <button @onclick="ChangeValue">Change from Child</button>
    </p>
```

```
    <NestedGrandchild @bind-GrandchildMessage:get="ChildMessage"
        @bind-GrandchildMessage:set="ChildMessageChanged" />
</div>

@code {
    [Parameter]
    public string? ChildMessage { get; set; }

    [Parameter]
    public EventCallback<string?> ChildMessageChanged { get; set; }

    private async Task ChangeValue()
    {
        await ChildMessageChanged.InvokeAsync(
            $"Set in Child {DateTime.Now}");
    }
}
```

`NestedGrandchild.razor`:

razor

```
<div class="border rounded m-1 p-1">
    <h3>Grandchild Component</h3>

    <p>Grandchild Message: <b>@GrandchildMessage</b></p>

    <p>
        <button @onclick="ChangeValue">Change from Grandchild</button>
    </p>
</div>

@code {
    [Parameter]
    public string? GrandchildMessage { get; set; }

    [Parameter]
    public EventCallback<string> GrandchildMessageChanged { get; set; }

    private async Task ChangeValue()
    {
        await GrandchildMessageChanged.InvokeAsync(
            $"Set in Grandchild {DateTime.Now}");
    }
}
```

For an alternative approach suited to sharing data in memory and across components that aren't necessarily nested, see ASP.NET Core Blazor state management.

# Bound field or property expression tree

To facilitate deeper interactions with a binding, Blazor allows you to capture of the expression tree of a bound field or property. This is achieved by defining a property with the field or property name suffixed with `Expression`. For any given field or property named `{FIELD OR PROPERTY NAME}`, the corresponding expression tree property is named `{FIELD OR PROPERTY NAME}Expression`.

The following `ChildParameterExpression` component identifies the `Year` expression's model and field name. A FieldIdentifier, which is used to obtain the model and field name, uniquely identifies a single field that can be edited. This may correspond to a property on a model object or can be any other named value. Use of a parameter's expression is useful when creating custom validation components, which isn't covered by the Microsoft Blazor documentation but is addressed by numerous third-party resources.

`ChildParameterExpression.razor`:

razor

```razor
@using System.Linq.Expressions

<ul>
    <li>Year model: @yearField.Model</li>
    <li>Year field name: @yearField.FieldName</li>
</ul>

@code {
    private FieldIdentifier yearField;

    [Parameter]
    public int Year { get; set; }

    [Parameter]
    public EventCallback<int> YearChanged { get; set; }

    [Parameter]
    public Expression<Func<int>> YearExpression { get; set; } = default!;

    protected override void OnInitialized()
    {
        yearField = FieldIdentifier.Create(YearExpression);
    }
}
```

`Parent3.razor`:

razor

```razor
@page "/parent-3"

<PageTitle>Parent 3</PageTitle>

<h1>Parent Example 3</h1>

<p>Parent <code>year</code>: @year</p>

<ChildParameterExpression @bind-Year="year" />

@code {
    private int year = 1979;
}
```

# Additional resources

- Parameter change detection and additional guidance on Razor component rendering
- ASP.NET Core Blazor forms overview

- Binding to radio buttons in a form
- Binding InputSelect options to C# object null values
- ASP.NET Core Blazor event handling: EventCallback section
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)