1. What Is Event Handling in Blazor?

Event handling is the mechanism by which a Blazor component reacts to **user interactions** such as:

- Mouse clicks
- Keyboard input
- Form submission
- Focus/blur
- Change events

Blazor uses **Razor syntax** to bind DOM events directly to **C# methods**, eliminating JavaScript for most UI logic.

2. How Event Handling Works Internally (High Level)

1. User interacts with a DOM element (e.g., clicks a button)
2. Browser raises a DOM event
3. Blazor intercepts the event
4. Event is dispatched to the component
5. Bound C# method executes
6. Component state changes
7. Blazor triggers **re-render (diff + DOM patch)**

No full page reload occurs.

Hooking into event delegates

In Blazor, UI events (such as onclick, onchange, onsubmit) are handled by delegates.
When you "hook into an event delegate," you are assigning a method (or lambda) to an event, so that Blazor invokes that method when the event occurs.

At runtime, Blazor:

1. **Captures the browser event**
2. **Maps it to a .NET event delegate**
3. **Invokes your C# method**

1. Basic Hooking: Method Group to Delegate

This is the simplest and most common way.

```
<button @onclick="HandleClick">Click Me</button>

@code {
    void HandleClick()
    {
        Console.WriteLine("Button clicked");
    }
}
```

What happens internally

- **@onclick expects a delegate of type EventCallback<MouseEventArgs>**
- **HandleClick matches the delegate signature**
- **Blazor hooks HandleClick into the event delegate**

Hooking Using Lambda Expressions

Used when:

- **You want to pass parameters**
- **You want inline logic**

```
<button @onclick="() => UpdateStatus("Approved")">Approve</button>

@code {
    string Status = "Pending";

    void UpdateStatus(string newStatus)
    {
        Status = newStatus;
    }
}
```

Here:

- **The lambda () => UpdateStatus("Approved") is converted into a delegate**
- **That delegate is hooked into the onclick event**

3. Hooking with Event Arguments

Some events provide event argument data.

```
<input @onchange="HandleChange" />

@code {
    void HandleChange(ChangeEventArgs e)
    {
        Console.WriteLine($"New value: {e.Value}");
    }
}
```

Delegate signature:

void HandleChange(ChangeEventArgs e)

Blazor ensures:

- **Browser event → ChangeEventArgs**
- **Delegate is invoked with populated data**

3. Basic Event Binding – @onclick

Demo 1: Button Click Counter

```
<h3>Counter</h3>

<p>Current Count: @count</p>

<button @onclick="Increment">Click Me</button>

@code {
    int count = 0;

    void Increment()
    {
        count++;
    }
}
```

Key Points

- @onclick binds the DOM click event
- Method can be void, Task, or async Task
- State change automatically triggers UI refresh

---

4. Inline Lambda Event Handlers

Used when logic is **simple or contextual**.

Demo 2: Inline Handler

```
<button @onclick="() => count += 5">
    Increment by 5
</button>
```

When to Use

- One-line logic
- No reuse needed

When NOT to Use

- Complex logic
- Reusability required

6. Handling Input Events (@onchange, @oninput)

6.1 @onchange (Default)

Triggered when focus leaves the input.

```
<input @onchange="OnNameChanged" />
<p>@name</p>

@code {
    string name = "";

    void OnNameChanged(ChangeEventArgs e)
    {
        name = e.Value?.ToString();
    }
}
```

6.2 @oninput (Real-time)

Triggered on every keystroke.

```
<input @oninput="OnTyping" />
<p>@text</p>

@code {
    string text = "";

    void OnTyping(ChangeEventArgs e)
    {
        text = e.Value?.ToString();
    }
}
```

7. Keyboard Events

Demo 4: Detect Enter Key

```
<input @onkeydown="HandleKeyDown" />

@code {
    void HandleKeyDown(KeyboardEventArgs e)
    {
        if (e.Key == "Enter")
        {
            Console.WriteLine("Enter pressed");
        }
    }
}
```

Common Keyboard Events

Event Purpose

@onkeydown Key pressed

@onkeyup Key released

@onkeypress Character input

9. Event Modifiers

Blazor provides **event modifiers** to control behavior.

9.1 @onclick:preventDefault

@onclick:preventDefault stops the browser's default behavior for an element after the click occurs, while still allowing the Blazor event handler to run.

This is equivalent to JavaScript:

event.preventDefault();

In Blazor, it is expressed declaratively:

@onclick:preventDefault

```
<a href="https://example.com"
   @onclick="HandleClick"
   @onclick:preventDefault>
   Click
</a>
```

Prevents navigation.

3. Example 1: Prevent Link Navigation

Problem

Clicking a link navigates away immediately.

```
<a href="https://example.com"
   @onclick="HandleClick">
   Click Me
</a>
```

Solution

```
<a href="https://example.com"
   @onclick="HandleClick"
   @onclick:preventDefault>
   Click Me
</a>

<p>@message</p>

@code {
    string message = "";

    void HandleClick()
    {
        message = "Link clicked, but navigation prevented.";
    }
}
```

Result

- Click event fires
- Navigation does **not** occur
- Page remains intact

---

4. Example 2: Prevent Form Submission

Problem

Form reloads the page on submit.

<form>

  <button type="submit" @onclick="Save">Save</button>

</form>

Solution

```
<form>
    <button type="submit"
            @onclick="Save"
            @onclick:preventDefault>
        Save
    </button>
</form>

@code {
    void Save()
    {
        Console.WriteLine("Saved without page reload");
    }
}
```

9.2 @onclick:stopPropagation

What is stopPropagation?

@onclick:stopPropagation stops an event from bubbling up the DOM tree.

It is the Blazor equivalent of JavaScript:

event.stopPropagation();

It does NOT:

- **Prevent the browser's default action**

- **Stop the Blazor handler itself**

It ONLY stops the event from reaching parent elements.

---

2. Why Event Bubbling Matters

In the browser:

- **Events start at the target element**
- **Then bubble upward to parent elements**

Example:

button → div → body → document

Without stopping propagation:

- **Clicking a button triggers all parent handlers**

```
<div @onclick="ParentClick">
    <button @onclick="ChildClick"
            @onclick:stopPropagation>
        Click
    </button>
</div>
```

Stops event bubbling.

---

3. Basic Example (Problem)

Without stopPropagation

```
<div @onclick="ParentClicked"
     style="padding:20px; border:2px solid black">

    <button @onclick="ChildClicked">
        Child Button
    </button>
</div>

@code {
    void ParentClicked()
    {
        Console.WriteLine("Parent clicked");
    }

    void ChildClicked()
    {
        Console.WriteLine("Child clicked");
    }
}
```

Result (Click Button)

Child clicked

Parent clicked

---

4. Solution: @onclick:stopPropagation

```
<div @onclick="ParentClicked"
     style="padding:20px; border:2px solid black">

    <button @onclick="ChildClicked"
            @onclick:stopPropagation>
        Child Button
    </button>
</div>
```

Result (Click Button)

Child clicked

Parent handler is not executed.

---

5. Real-World Use Case: Card with Action Buttons

Problem

Clicking Delete should not open the card.

```
<div class="card" @onclick="OpenDetails">
   <h4>Order #123</h4>

   <button @onclick="Delete">
      Delete
   </button>
</div>
```

Clicking Delete:

- **Deletes the item**
- **Also opens details (undesired)**

Correct Implementation

```
<div class="card" @onclick="OpenDetails">
   <h4>Order #123</h4>

   <button @onclick="Delete"
           @onclick:stopPropagation>
      Delete
   </button>
</div>
```

```
@code {
    void OpenDetails()
    {
        Console.WriteLine("Opening details...");
    }

    void Delete()
    {
        Console.WriteLine("Deleting...");
    }
}
```

When You SHOULD Use stopPropagation

Use it when:

- **Child actions exist inside clickable containers**
- **You want action buttons inside cards, lists, rows**
- **Context menus inside rows**
- **Modal close buttons inside overlays**

stopPropagation vs preventDefault

Feature	stopPropagation	preventDefault
Stops bubbling	✅ Yes	❌ No
Stops default action	❌ No	✅ Yes
Allows handler	✅ Yes	✅ Yes

Combined Usage

```
<button @onclick="Click"
        @onclick:stopPropagation
        @onclick:preventDefault>
    Click
</button>
```

Async Event Handlers

```
<button @onclick="LoadData">Load</button>

<p>@message</p>

@code {
    string message = "";

    async Task LoadData()
    {
        message = "Loading...";
        await Task.Delay(2000);
        message = "Completed";
    }
}
```

Best Practice

- Always use async Task
- Avoid async void