
Debugging the Code in Blazor

Debugging is the systematic process of **finding, understanding, and fixing defects** in an application. In Blazor, debugging differs slightly depending on **hosting model**:

- **Blazor Server** → runs on the server (.NET runtime on IIS/Kestrel)
- **Blazor WebAssembly (WASM)** → runs in the browser (Mono/.NET runtime in WebAssembly)

We will cover both in detail.

---

1. Making Things Break (Intentional Failure)

Before learning debugging, you must **intentionally break code** to observe:

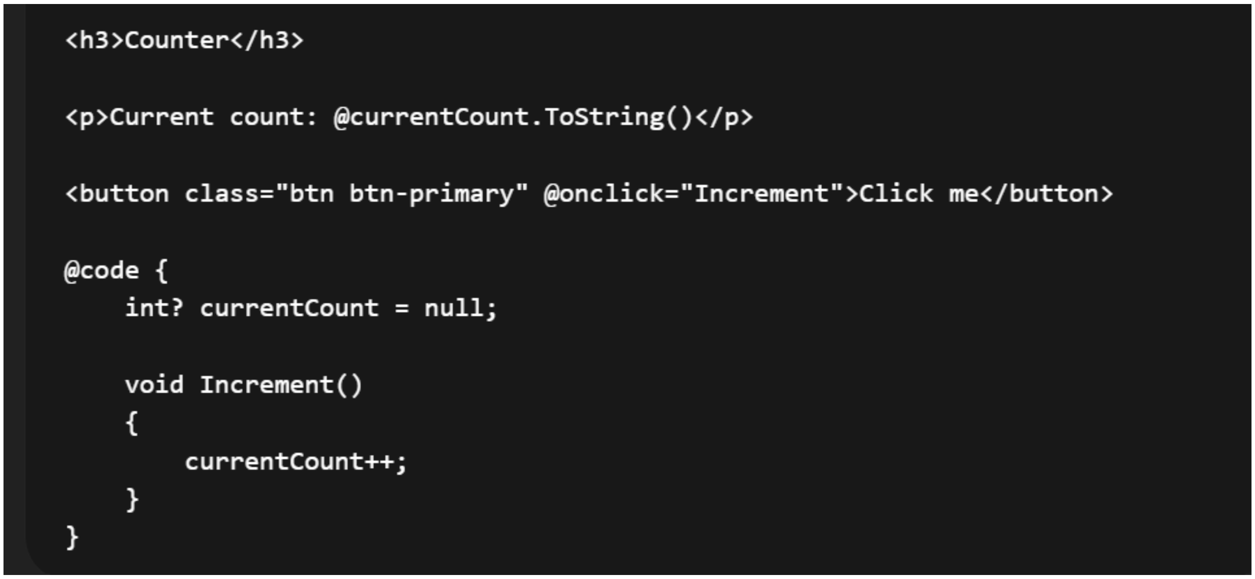
- Where the failure occurs
- How the debugger behaves
- What tools are available

Common Ways to Break Code in Blazor

1. **NullReferenceException**
2. **Logic bugs (wrong output)**
3. **Async timing issues**
4. **UI not updating**
5. **JavaScript interop failures**

Example 1: NullReferenceException

Component: Counter.razor



```
<h3>Counter</h3>

<p>Current count: @currentCount.ToString()</p>

<button class="btn btn-primary" @onclick="Increment">Click me</button>

@code {
    int? currentCount = null;

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

What breaks?

- currentCount.ToString() throws NullReferenceException

This is ideal for learning **breakpoints and stack traces**.

---

2. Debugging Blazor Server

How Blazor Server Works (Debug Perspective)

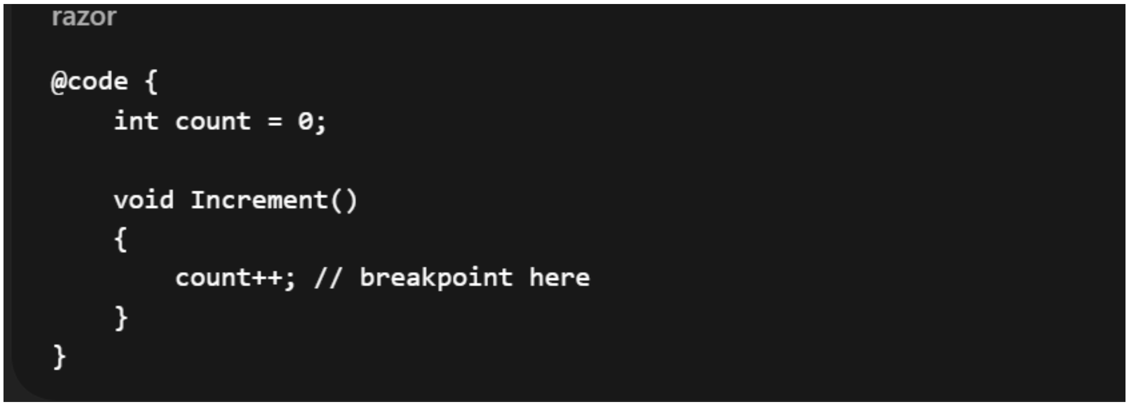
- Code executes **on the server**
- UI updates sent via **SignalR**
- Debugging is almost identical to ASP.NET Core MVC

Debugging Setup (Blazor Server)

1. Open project in **Visual Studio**
2. Select **Debug**
3. Press **F5**
4. Browser launches automatically

Using Breakpoints

Set a breakpoint inside a component:



razor

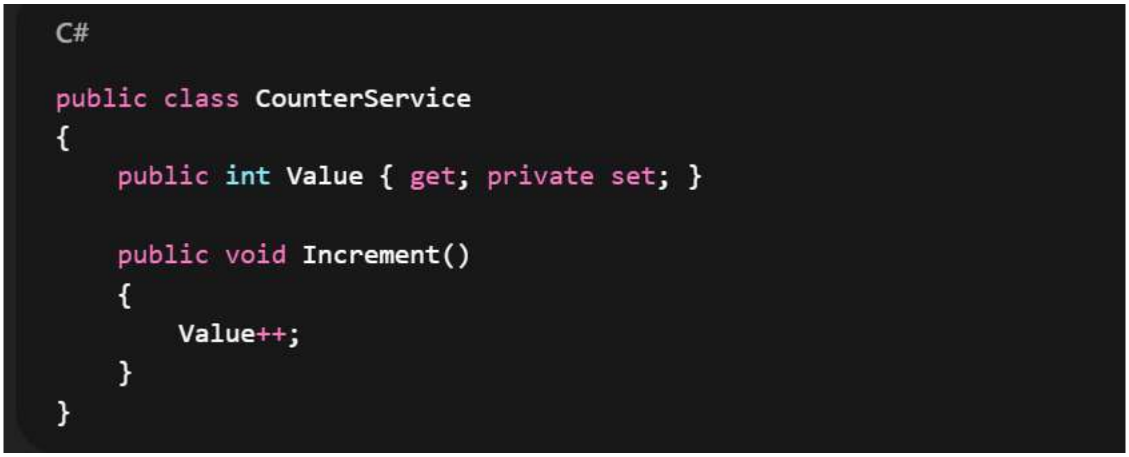
```
@code {
    int count = 0;

    void Increment()
    {
        count++; // breakpoint here
    }
}
```

What You Can Inspect

- Local variables
- Component state
- Call stack
- Dependency-injected services

Example: Debugging a Service Issue

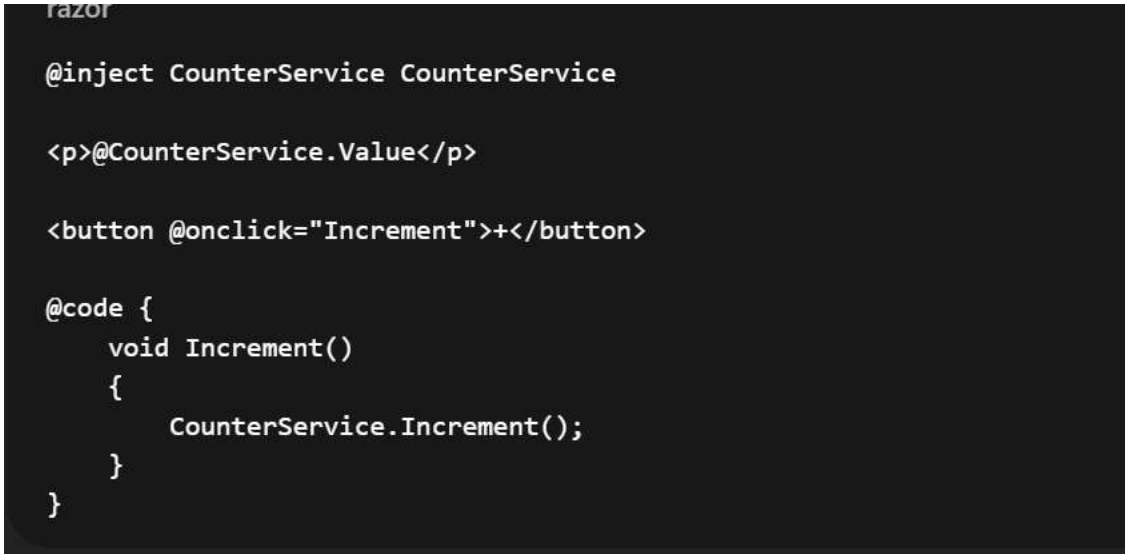
Service



C#

```
public class CounterService
{
    public int Value { get; private set; }

    public void Increment()
    {
        Value++;
    }
}
```

Component



```
razor

@inject CounterService CounterService

<p>@CounterService.Value</p>

<button @onclick="Increment">+</button>

@code {
    void Increment()
    {
        CounterService.Increment();
    }
}
```

Debugging Focus

- Is the service registered as Singleton?
- Is state persisting across requests?

Common Blazor Server Debugging Issues

Issue	Cause
UI freezes	SignalR connection lost
State resets	Incorrect DI lifetime
Multiple users sharing state	Singleton misuse
JS interop fails	Prerendering issue

---

3. Debugging Blazor WebAssembly (WASM)

Key Difference

Blazor Server	**Blazor WASM**
Runs on server	Runs in browser
.NET CLR	Mono WASM runtime
Easy debugging	More complex debugging

Debugging in Visual Studio

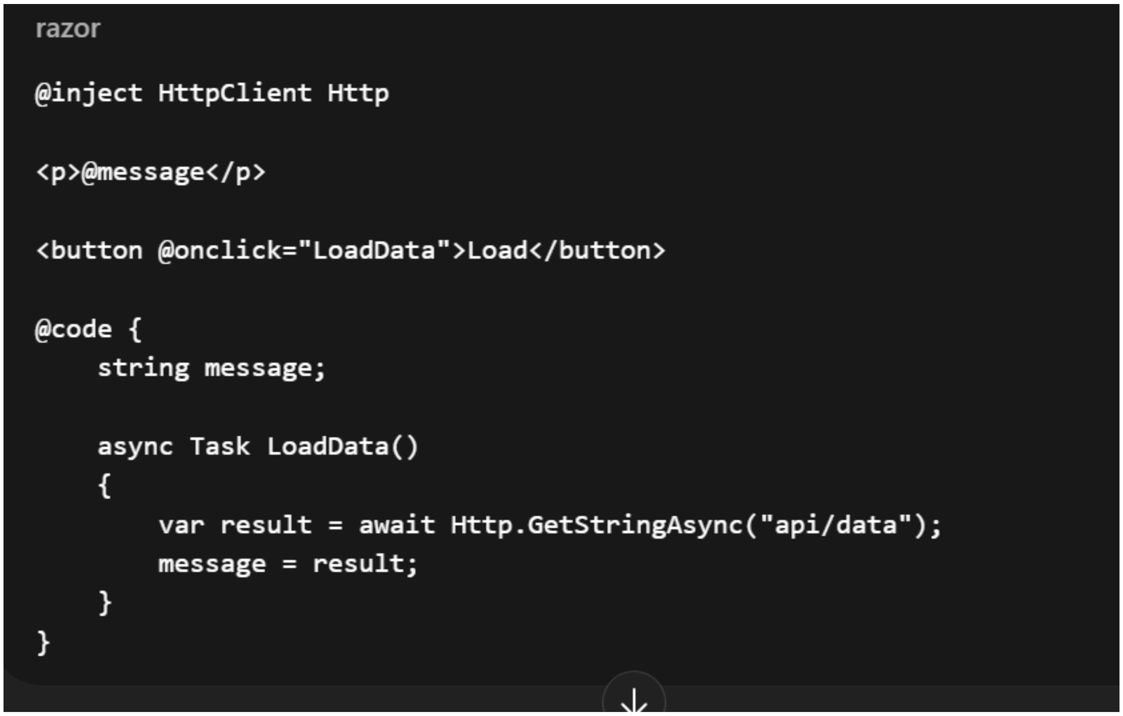
1. Set **startup project**
2. Run with **Debug**
3. Browser launches
4. Debugger attaches to WASM runtime

You can:

- Set breakpoints in .razor and .cs
- Inspect variables
- Step through code

⚠️ Debugging may feel slower due to WebAssembly execution.

Example: Debugging API Call



```razor
@inject HttpClient Http

<p>@message</p>

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

@code {
    string message;

    async Task LoadData()
    {
        var result = await Http.GetStringAsync("api/data");
        message = result;
    }
}
```

Breakpoints

- Before API call
- After response
- On exception

Common WASM Debugging Problems

Problem	**Explanation**
Breakpoint not hit	Browser debugger not attached
Null values	Async lifecycle timing
Slow debugging	WASM runtime overhead
API fails	CORS / HTTPS issues

---

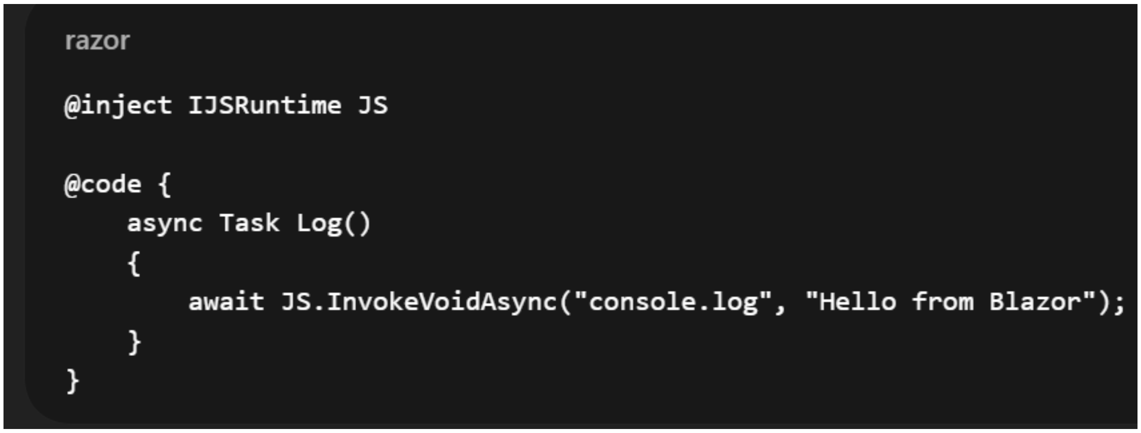
4. Debugging Blazor WebAssembly in the Web Browser

This is **critical for instructors**.

Chrome / Edge DevTools

Press **F12**

Console Debugging



razor

```
@inject IJSRuntime JS

@code {
    async Task Log()
    {
        await JS.InvokeVoidAsync("console.log", "Hello from Blazor");
    }
}
```

Use **Console tab** to:

- View logs
- Catch JS errors
- Inspect network calls

Network Tab (Very Important)

Used to debug:

- API failures
- HTTP status codes
- Payload issues

Steps:

1. Open DevTools
2. Go to **Network**
3. Trigger API call
4. Inspect request/response

Source Tab (Advanced)

- View compiled WASM files
- Inspect JS interop files
- Debug JavaScript isolation modules

5. Hot Reload (Blazor Productivity Feature)

What Is Hot Reload?

Hot Reload allows you to:

- Modify code
- Save file
- See UI changes **without restarting the app**

What Hot Reload Supports

Change Type	**Supported**
HTML markup	Yes
CSS	Yes
Simple C# logic	Yes
Method signature	No
DI registration	No

Example: Hot Reload in Action

Change markup:

```
<h3 style="color:red">Counter</h3>
```

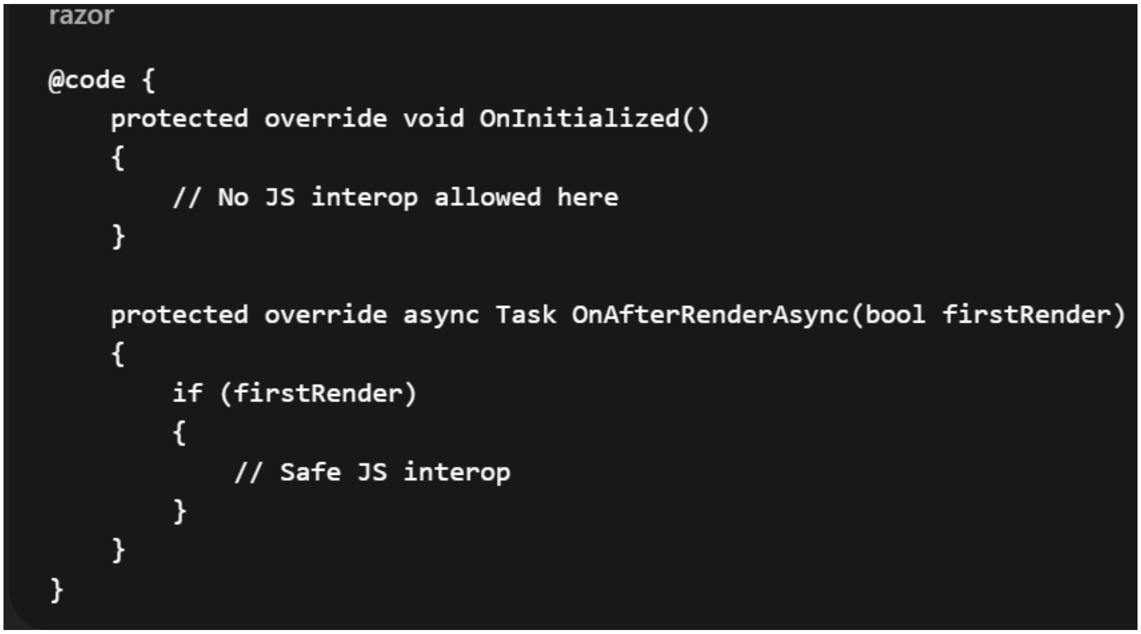
Save file → UI updates instantly.

Hot Reload Limitations (Important for Exams & Interviews)

- Cannot add new classes
- Cannot change method signatures
- Sometimes requires restart for state reset

6. Debugging Lifecycle Issues (Very Common)

Example: OnInitialized vs OnAfterRender



```
@code {
    protected override void OnInitialized()
    {
        // No JS interop allowed here
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            // Safe JS interop
        }
    }
}
```

Common Error

JavaScript interop calls cannot be issued at this time

Debugging Insight

- Check lifecycle method
- Verify render mode

Blazor Debugging Comparison

Feature	**Server**	**WASM**
Breakpoints	Excellent	Good
Browser tools	Limited	Essential
Performance	Fast	Slower
Hot Reload	Stable	Improving

8. Real-World Debugging Checklist

- Set breakpoints early
- Inspect DI lifetimes
- Check browser console
- Verify API endpoints
- Restart app if Hot Reload fails
- Understand lifecycle methods