# JavaScript Interop in Blazor (.NET 10.0)

What is JavaScript Interop?

**JavaScript Interop (JS Interop)** is the mechanism that allows:

- **.NET (Blazor) → JavaScript**
- **JavaScript → .NET**

Blazor runs on top of the browser. The browser understands **JavaScript**, not .NET.
Interop is the **bridge** between Blazor and browser APIs.

1. Why Do We Need JavaScript in Blazor?

Blazor is powerful, but **not everything is available in .NET**.

Common Scenarios Where JavaScript Is Required

| **Requirement** | **Reason** |
|---|---|
| DOM manipulation | Browser APIs are JavaScript-based |
| Access browser APIs | Clipboard, Geolocation, LocalStorage |
| UI libraries | Chart.js, Bootstrap JS, jQuery plugins |
| Performance-critical UI | Canvas, WebGL |
| Legacy JS libraries | Existing enterprise JS code |

**Key Principle**

**Blazor handles application logic**
**JavaScript handles browser-specific behavior**

**2. .NET → JavaScript (Calling JS from Blazor)**

This is the **most common scenario**.

**Step 1: Inject IJSRuntime**

```
@inject IJSRuntime JS
```

### Step 2: Call JavaScript

```
await JS.InvokeVoidAsync("alert", "Hello from Blazor!");
```

## 3. Global JavaScript (The Old Way)

### What Is Global JavaScript?

JavaScript functions defined in wwwroot/*.js and attached to window.

### Example: wwwroot/js/site.js

```
window.showMessage = function (message) {
    alert(message);
};
```

### Register Script (App.razor or MainLayout.razor)

```
<script src="js/site.js"></script>
```

### Call from Blazor

```
<button @onclick="ShowMessage">Show</button>

@code {
    async Task ShowMessage()
    {
        await JS.InvokeVoidAsync("showMessage", "Hello Global JS");
    }
}
```

### ❌ Problems with Global JS

- Pollutes window
- No scoping
- Hard to maintain
- Naming conflicts

---

**4. JavaScript Isolation (Recommended Way)**

**JavaScript Isolation** scopes JavaScript to a component.

**Folder Structure**

```
/Components
  /InteropDemo.razor
  /InteropDemo.razor.js
```

**InteropDemo.razor.js**

```
export function showMessage(message) {
    alert(message);
}
```

**InteropDemo.razor**

```
@inject IJSRuntime JS

<button @onclick="Show">Show Isolated JS</button>

@code {
    private IJSObjectReference? _module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            _module = await JS.InvokeAsync<IJSObjectReference>(
                "import", "./Components/InteropDemo.razor.js");
        }
    }

    async Task Show()
    {
        await _module!.InvokeVoidAsync("showMessage", "Hello Isolated JS");
    }
}
```

✅ **Advantages**

- Scoped
- Tree-shakable
- Safer
- Component-based

---

### 5. JavaScript → .NET (Calling .NET from JS)

**Two Types**

1. **Static .NET method**
2. **Instance .NET method**

---

### 6. JavaScript → .NET (Static Method Call)

**Step 1: Static Method in .NET**

```
using Microsoft.JSInterop;

public class JsCallbacks
{
    [JSInvokable]
    public static void Notify(string message)
    {
        Console.WriteLine($"JS says: {message}");
    }
}
```

---

### Step 2: JavaScript Call

```
DotNet.invokeMethodAsync(
    'YourProjectName',
    'Notify',
    'Hello from JS'
);
```

### Characteristics

| Aspect | Static Call |
|---|---|
| Lifetime | Application-level |
| State | ❌ No instance state |
| Use case | Logging, notifications |

---

## 7. JavaScript → .NET (Instance Method Call)

Used when **component state matters**.

---

### Step 1: Component Code

```
@inject IJSRuntime JS

<button @onclick="Register">Register Callback</button>
```

```
@code {
    DotNetObjectReference<JsInstanceDemo>? _objRef;

    void Register()
    {
        _objRef = DotNetObjectReference.Create(this);
        JS.InvokeVoidAsync("registerDotNet", _objRef);
    }

    [JSInvokable]
    public void ReceiveMessage(string msg)
    {
        Console.WriteLine($"Received: {msg}");
    }

    public void Dispose()
    {
        _objRef?.Dispose();
    }
}
```

---

**Step 2: JavaScript**

```
window.registerDotNet = function (dotNetRef) {
    dotNetRef.invokeMethodAsync("ReceiveMessage", "Hello Instance!");
};
```

**Characteristics**

| Aspect | Instance Call |
|---|---|
| Lifetime | Component-based |
| State | ✅ Yes |
| Use case | UI updates, events |

---

**8. Implementing an Existing JavaScript Library**

**Example: Using Chart.js**

---

**Step 1: Add JS Library**

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

---

**Step 2: Wrapper JS**

```
window.createChart = function (canvasId) {
  const ctx = document.getElementById(canvasId);
  new Chart(ctx, {
    type: 'bar',
    data: {
      labels: ['A', 'B', 'C'],
      datasets: [{
        label: 'Sales',
        data: [10, 20, 30]
      }]
    }
  });
};
```

---

**Step 3: Blazor Component**

```
@inject IJSRuntime JS


<canvas id="myChart"></canvas>


@code {
    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            await JS.InvokeVoidAsync("createChart", "myChart");
        }
    }
}
```

---

**9. JavaScript Interop in WebAssembly**

Blazor WebAssembly runs **entirely in the browser**.

**Key Differences**

| Feature | Server | WebAssembly |
|---|---|---|
| Latency | Network | Local |
| JS Interop | SignalR | Direct |
| Performance | Slower | Faster |
| Offline | ❌ | ✅ |

---

### 10. WebAssembly: .NET → JavaScript

Same API:

await JS.InvokeVoidAsync("alert", "Hello WASM");

But execution is **direct** in browser memory.

---

### 11. WebAssembly: JavaScript → .NET

Works identically:

DotNet.invokeMethodAsync("AppName", "Notify");

or instance reference.

**Summary Table**

| **Scenario** | **Technique** |
|---|---|
| Browser API | JS Interop |
| Component JS | JS Isolation |
| JS → .NET | [JSInvokable] |
| Stateful callback | Instance method |
| UI library | Wrapper JS |
| WASM performance | Direct JS calls |