

razor

```
<EditForm ... EditContext="editContext" ...>
    ...
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??= new();
        editContext = new(Model);
    }
}
```

Assign **either** an [EditContext](#) or a [Model](#) to an [EditForm](#). If both are assigned, a runtime error is thrown.

Supported types

Binding supports:

- Primitive types
- Collections
- Complex types
- Recursive types
- Types with constructors
- Enums

You can also use the [\[DataMember\]](#) and [\[IgnoreDataMember\]](#) attributes to customize model binding. Use these attributes to rename properties, ignore properties, and mark properties as required.

Additional binding options

Additional model binding options are available from [RazorComponentsServiceOptions](#) when calling [AddRazorComponents](#):

- [MaxFormMappingCollectionSize](#): Maximum number of elements allowed in a form collection.

- [MaxFormMappingRecursionDepth](#): Maximum depth allowed when recursively mapping form data.
- [MaxFormMappingErrorCount](#): Maximum number of errors allowed when mapping form data.
- [MaxFormMappingKeySize](#): Maximum size of the buffer used to read form data keys.

The following demonstrates the default values assigned by the framework:

C#

```
builder.Services.AddRazorComponents(options =>
{
    options.FormMappingUseCurrentCulture = true;
    options.MaxFormMappingCollectionSize = 1024;
    options.MaxFormMappingErrorCount = 200;
    options.MaxFormMappingKeySize = 1024 * 2;
    options.MaxFormMappingRecursionDepth = 64;
}).AddInteractiveServerComponents();
```

Form names

Use the [FormName](#) parameter to assign a form name. Form names must be unique to bind model data. The following form is named `RomulanAle`:

razor

```
<EditForm ... FormName="RomulanAle" ...>
...
</EditForm>
```

Supplying a form name:

- Is required for all forms that are submitted by statically-rendered server-side components.
- Isn't required for forms that are submitted by interactively-rendered components, which includes forms in Blazor WebAssembly apps and components with an interactive render mode. However, we recommend supplying a unique form name for every form to prevent runtime form posting errors if interactivity is ever dropped for a form.

The form name is only checked when the form is posted to an endpoint as a traditional HTTP POST request from a statically-rendered server-side component. The framework

doesn't throw an exception at the point of rendering a form, but only at the point that an HTTP POST arrives and doesn't specify a form name.

There's an unnamed (empty string) form scope above the app's root component, which suffices when there are no form name collisions in the app. If form name collisions are possible, such as when including a form from a library and you have no control of the form name used by the library's developer, provide a form name scope with the [FormMappingScope](#) component in the Blazor Web App's main project.

In the following example, the `HelloFormFromLibrary` component has a form named `Hello` and is in a library.

`HelloFormFromLibrary.razor`:

razor

```
<EditForm FormName="Hello" Model="this" OnSubmit="Submit">
    <InputText @bind-Value="Name" />
    <button type="submit">Submit</button>
</EditForm>

@if (submitted)
{
    <p>Hello @Name from the library's form!</p>
}

@code {
    bool submitted = false;

    [SupplyParameterFromForm]
    private string? Name { get; set; }

    private void Submit() => submitted = true;
}
```

The following `NamedFormsWithScope` component uses the library's `HelloFormFromLibrary` component and also has a form named `Hello`. The [FormMappingScope](#) component's scope name is `ParentContext` for any forms supplied by the `HelloFormFromLibrary` component. Although both of the forms in this example have the form name (`Hello`), the form names don't collide and events are routed to the correct form for form POST events.

`NamedFormsWithScope.razor`:

razor

```

@page "/named-forms-with-scope"

<div>Hello form from a library</div>

<FormMappingScope Name="ParentContext">
    <HelloFormFromLibrary />
</FormMappingScope>

<div>Hello form using the same form name</div>

<EditForm FormName="Hello" Model="this" OnSubmit="Submit">
    <InputText @bind-Value="Name" />
    <button type="submit">Submit</button>
</EditForm>

@if (submitted)
{
    <p>Hello @Name from the app form!</p>
}

@code {
    bool submitted = false;

    [SupplyParameterFromForm]
    private string? Name { get; set; }

    private void Submit() => submitted = true;
}

```

Supply a parameter from the form ([SupplyParameterFromForm])

The `[SupplyParameterFromForm]` attribute indicates that the value of the associated property should be supplied from the form data for the form. Data in the request that matches the name of the property is bound to the property. Inputs based on `InputBase<TValue>` generate form value names that match the names Blazor uses for model binding. Unlike component parameter properties (`[Parameter]`), properties annotated with `[SupplyParameterFromForm]` aren't required to be marked `public`.

You can specify the following form binding parameters to the `[SupplyParameterFromForm]` attribute:

- **Name:** Gets or sets the name for the parameter. The name is used to determine the prefix to use to match the form data and decide whether or not the value needs to be bound.

- **FormName**: Gets or sets the name for the handler. The name is used to match the parameter to the form by form name to decide whether or not the value needs to be bound.

The following example independently binds two forms to their models by form name.

Starship6.razor:

razor

```
@page "/starship-6"
@inject ILogger<Starship6> Logger

<EditForm Model="Model1" OnSubmit="Submit1" FormName="Holodeck1">
    <div>
        <label>
            Holodeck 1 Identifier:
            <InputText @bind-Value="Model1!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

<EditForm Model="Model2" OnSubmit="Submit2" FormName="Holodeck2">
    <div>
        <label>
            Holodeck 2 Identifier:
            <InputText @bind-Value="Model2!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm(FormName = "Holodeck1")]
    private Holodeck? Model1 { get; set; }

    [SupplyParameterFromForm(FormName = "Holodeck2")]
    private Holodeck? Model2 { get; set; }

    protected override void OnInitialized()
    {
        Model1 ??= new();
        Model2 ??= new();
    }

    private void Submit1() => Logger.LogInformation("Submit1: Id={Id}",
        Model1?.Id);
}
```

```
private void Submit2() => Logger.LogInformation("Submit2: Id={Id}",
Model2?.Id);

public class Holodeck
{
    public string? Id { get; set; }
}
}
```

Nest and bind forms

The following guidance demonstrates how to nest and bind child forms.

The following ship details class (`ShipDetails`) holds a description and length for a subform.

`ShipDetails.cs`:

C#

```
namespace BlazorSample;

public class ShipDetails
{
    public string? Description { get; set; }
    public int? Length { get; set; }
}
```

The following `Ship` class names an identifier (`Id`) and includes the ship details.

`Ship.cs`:

C#

```
namespace BlazorSample
{
    public class Ship
    {
        public string? Id { get; set; }
        public ShipDetails Details { get; set; } = new();
    }
}
```

The following subform is used for editing values of the `ShipDetails` type. This is implemented by inheriting `Editor<T>` at the top of the component. `Editor<T>` ensures

that the child component generates the correct form field names based on the model (T), where T in the following example is ShipDetails.

StarshipSubform.razor:

```
razor

@inherits Editor<ShipDetails>

<div>
    <label>
        Description:
        <InputText @bind-Value="Value!.Description" />
    </label>
</div>
<div>
    <label>
        Length:
        <InputNumber @bind-Value="Value!.Length" />
    </label>
</div>
```

The main form is bound to the Ship class. The StarshipSubform component is used to edit ship details, bound as Model!.Details.

Starship7.razor:

```
razor

@page "/starship-7"
@inject ILogger<Starship7> Logger

<EditForm Model="Model" OnSubmit="Submit" FormName="Starship7">
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <StarshipSubform @bind-Value="Model!.Details" />
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Ship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();
}
```

```

private void Submit() =>
    Logger.LogInformation("Id = {Id} Desc = {Description} Length = {Length}",
        Model?.Id, Model?.Details?.Description, Model?.Details?.Length);
}

```

Initialize form data with static SSR

When a component adopts static SSR, the [OnInitialized{Async} lifecycle method](#) and the [OnParametersSet{Async} lifecycle method](#) fire when the component is initially rendered and on every form POST to the server. To initialize form model values, confirm if the model already has data before assigning new model values in `OnParametersSet{Async}`, as the following example demonstrates.

StarshipInit.razor:

razor

```

@page "/starship-init"
@Inject ILogger<StarshipInit> Logger

<EditForm Model="Model" OnValidSubmit="Submit" FormName="StarshipInit">
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    protected override void OnParametersSet()
    {
        if (Model!.Id == default)
        {
            LoadData();
        }
    }

    private void LoadData()
    {
        Model!.Id = "Set by LoadData";
    }
}

```



```

    }

    private void Submit()
    {
        Logger.LogInformation("Id = {Id}", Model?.Id);
    }

    public class Starship
    {
        public string? Id { get; set; }
    }
}

```

Advanced form mapping error scenarios

The framework instantiates and populates the [FormMappingContext](#) for a form, which is the context associated with a given form's mapping operation. Each mapping scope (defined by a [FormMappingScope](#) component) instantiates [FormMappingContext](#). Each time a `[SupplyParameterFromForm]` asks the context for a value, the framework populates the [FormMappingContext](#) with the attempted value and any mapping errors.

Developers aren't expected to interact with [FormMappingContext](#) directly, as it's mainly a source of data for [InputBase<TValue>](#), [EditContext](#), and other internal implementations to show mapping errors as validation errors. In advanced custom scenarios, developers can access [FormMappingContext](#) directly as a `[CascadingParameter]` to write custom code that consumes the attempted values and mapping errors.

Custom input components

For custom input processing scenarios, the following subsections demonstrate custom input components:

- [Input component based on InputBase<T>](#): The component inherits from [InputBase<TValue>](#), which provides a base implementation for binding, callbacks, and validation. Components that inherit from [InputBase<TValue>](#) must be used in a Blazor form ([EditForm](#)).
- [Input component with full developer control](#): The component takes full control of input processing. The component's code must manage binding, callbacks, and validation. The component can be used inside or outside of a Blazor form.

We recommend that you derive your custom input components from [InputBase<TValue>](#) unless specific requirements prevent you from doing so. The

`InputBase<TValue>` class is actively maintained by the ASP.NET Core team, ensuring it stays up-to-date with the latest Blazor features and framework changes.

Input component based on `InputBase<T>`

The following example component:

- Inherits from `InputBase<TValue>`. Components that inherit from `InputBase<TValue>` must be used in a Blazor form (`EditForm`).
- Takes boolean input from a checkbox.
- Sets the background color of its container `<div>` based on the checkbox's state, which occurs when the `AfterChange` method executes after binding (`@bind:after`).
- Is required to override the base class's `TryParseValueFromString` method but doesn't process string input data because a checkbox doesn't provide string data. Example implementations of `TryParseValueFromString` for other types of input components that process string input are available in the [ASP.NET Core reference source](#).

❗ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).

`EngineeringApprovalInputDerived.razor`:

razor

```
@using System.Diagnostics.CodeAnalysis
@inherits InputBase<bool>

<div class="@divCssClass">
    <label>
        Engineering Approval:
        <input @bind="CurrentValue" @bind:after="AfterChange"
class="@CssClass"
        type="checkbox" />
    </label>
</div>

@code {
    private string? divCssClass;
```

```

private void AfterChange()
{
    divCssClass = CurrentValue ? "bg-success text-white" : null;
}

protected override bool TryParseValueFromString(
    string? value, out bool result,
    [NotNullWhen(false)] out string? validationErrorMessage)
    => throw new NotSupportedException(
        "This component does not parse string inputs. " +
        $"Bind to the '{nameof(CurrentValue)}' property, " +
        $"not '{nameof(CurrentValueAsString)}'." );
}

```

To use the preceding component in the [starship example form](#) ([Starship3.razor/Starship.cs](#)), replace the `<div>` block for the engineering approval field with an `EngineeringApprovalInputDerived` component instance bound to the model's `IsValidatedDesign` property:

diff

```

- <div>
-     <label>
-         Engineering Approval:
-         <InputCheckbox @bind-Value="Model!.IsValidatedDesign" />
-     </label>
- </div>
+ <EngineeringApprovalInputDerived @bind-Value="Model!.IsValidatedDesign" />

```

Input component with full developer control

The following example component:

- Doesn't inherit from `InputBase<TValue>`. The component takes full control of input processing, including binding, callbacks, and validation. The component can be used inside or outside of a Blazor form ([EditForm](#)).
- Takes boolean input from a checkbox.
- Changes the background color if the checkbox is checked.

Code in the component includes:

- The `value` property is used with two-way binding to get or set the value of the input. `ValueChanged` is the callback that updates the bound value.
- When used in a Blazor form:
 - The `EditContext` is a [cascading value](#).

- `fieldCssClass` styles the field based on the result of `EditContext` validation.
- `ValueExpression` is an expression (`Expression<Func<T>>`) assigned by the framework that identifies the bound value.
- `FieldIdentifier` uniquely identifies a single field that can be edited, usually corresponding to a model property. The field identifier is created with the expression that identifies the bound value (`ValueExpression`).
- In the `OnChange` event handler:
 - The value of the checkbox input is obtained from `InputFileChangeEventArgs`.
 - The background color and text color of the container `<div>` element are set.
 - `EventCallback.InvokeAsync` invokes the delegate associated with the binding and dispatches an event notification to consumers that the value has changed.
 - If the component is used in an `EditForm` (the `EditContext` property isn't `null`), `EditContext.NotifyFieldChanged` is called to trigger validation.

`EngineeringApprovalInputStandalone.razor`:

razor

```
@using System.Globalization
@using System.Linq.Expressions

<div class="@divCssClass">
    <label>
        Engineering Approval:
        <input class="@fieldCssClass" @onchange="OnChange" type="checkbox"
            value="@Value" />
    </label>
</div>

@code {
    private string? divCssClass;
    private FieldIdentifier fieldIdentifier;
    private string? fieldCssClass =>
        EditContext?.FieldCssClass(fieldIdentifier);

    [CascadingParameter]
    private EditContext? EditContext { get; set; }

    [Parameter]
    public bool? Value { get; set; }

    [Parameter]
    public EventCallback<bool> ValueChanged { get; set; }

    [Parameter]
    public Expression<Func<bool>>? ValueExpression { get; set; }

    protected override void OnInitialized()
```

```

{
    fieldIdentifier = FieldIdentifier.Create(ValueExpression!);
}

private async Task OnChange(ChangeEventArgs args)
{
    BindConverter.TryConvertToBool(args.Value,
        CultureInfo.CurrentCulture,
        out var value);

    divCssClass = value ? "bg-success text-white" : null;

    await ValueChanged.InvokeAsync(value);
    EditContext?.NotifyFieldChanged(fieldIdentifier);
}
}

```

To use the preceding component in the [starship example form](#) ([Starship3.razor/Starship.cs](#)), replace the `<div>` block for the engineering approval field with a `EngineeringApprovalInputStandalone` component instance bound to the model's `IsValidatedDesign` property:

diff

```

- <div>
-     <label>
-         Engineering Approval:
-         <InputCheckbox @bind-Value="Model!.IsValidatedDesign" />
-     </label>
- </div>
+ <EngineeringApprovalInputStandalone @bind-Value="Model!.IsValidatedDesign"
/>

```

The `EngineeringApprovalInputStandalone` component is also functional outside of an [EditForm](#):

razor

```

<EngineeringApprovalInputStandalone @bind-Value="ValidDesign" />

<div>
    <b>ValidDesign:</b> @ValidDesign
</div>

@code {
    private bool ValidDesign { get; set; }
}

```

Radio buttons

The example in this section is based on the `Starfleet Starship Database` form (`Starship3` component) of the [Example form](#) section of this article.

Add the following [enum types](#) to the app. Create a new file to hold them or add them to the `Starship.cs` file.

C#

```
public class ComponentEnums
{
    public enum Manufacturer { SpaceX, NASA, ULA, VirginGalactic, Unknown }
    public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue,
VoyagerOrange }
    public enum Engine { Ion, Plasma, Fusion, Warp }
}
```

Make the `ComponentEnums` class accessible to the:

- `Starship` model in `Starship.cs` (for example, `using static ComponentEnums;`).
- `Starfleet Starship Database` form (`Starship3.razor`) (for example, `@using static ComponentEnums`).

Use `InputRadio<TValue>` components with the `InputRadioGroup<TValue>` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the [Example form](#) section of the *Input components* article:

C#

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a
manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Update the `Starfleet Starship Database` form (`Starship3` component) of the [Example form](#) section of the *Input components* article. Add the components to produce:

- A radio button group for the ship manufacturer.

- A nested radio button group for engine and ship color.

ⓘ Note

Nested radio button groups aren't often used in forms because they can result in a disorganized layout of form controls that may confuse users. However, there are cases when they make sense in UI design, such as in the following example that pairs recommendations for two user inputs, ship engine and ship color. One engine and one color are required by the form's validation. The form's layout uses nested `InputRadioGroup<TValue>`s to pair engine and color recommendations. However, the user can combine any engine with any color to submit the form.

ⓘ Note

Be sure to make the `ComponentEnums` class available to the component for the following example:

```
razor
```

```
@using static ComponentEnums
```

```
razor
```

```
<fieldset>
  <legend>Manufacturer</legend>
  <InputRadioGroup @bind-Value="Model!.Manufacturer">
    @foreach (var manufacturer in Enum.GetValues<Manufacturer>())
    {
      <div>
        <label>
          <InputRadio Value="manufacturer" />
          @manufacturer
        </label>
      </div>
    }
  </InputRadioGroup>
</fieldset>

<fieldset>
  <legend>Engine and Color</legend>
  <p>
    Engine and color pairs are recommended, but any
    combination of engine and color is allowed.
  </p>
  <InputRadioGroup Name="engine" @bind-Value="Model!.Engine">
    <InputRadioGroup Name="color" @bind-Value="Model!.Color">
```

```

<div style="margin-bottom:5px">
  <div>
    <label>
      <InputRadio Name="engine" Value="Engine.Ion" />
      Ion
    </label>
  </div>
  <div>
    <label>
      <InputRadio Name="color" Value="Color.ImperialRed"
/>
      Imperial Red
    </label>
  </div>
</div>
<div style="margin-bottom:5px">
  <div>
    <label>
      <InputRadio Name="engine" Value="Engine.Plasma" />
      Plasma
    </label>
  </div>
  <div>
    <label>
      <InputRadio Name="color"
Value="Color.SpacecruiserGreen" />
      Spacecruiser Green
    </label>
  </div>
</div>
<div style="margin-bottom:5px">
  <div>
    <label>
      <InputRadio Name="engine" Value="Engine.Fusion" />
      Fusion
    </label>
  </div>
  <div>
    <label>
      <InputRadio Name="color" Value="Color.StarshipBlue"
/>
      Starship Blue
    </label>
  </div>
</div>
<div style="margin-bottom:5px">
  <div>
    <label>
      <InputRadio Name="engine" Value="Engine.Warp" />
      Warp
    </label>
  </div>
  <div>
    <label>
      <InputRadio Name="color" Value="Color.VoyagerOrange"

```



```

/>
                Voyager Orange
            </label>
        </div>
    </div>
</InputRadioGroup>
</InputRadioGroup>
</fieldset>

```

ⓘ Note

If `Name` is omitted, `InputRadio<TValue>` components are grouped by their most recent ancestor.

If you implemented the preceding Razor markup in the `Starship3` component of the [Example form](#) section of the *Input components* article, update the logging for the `Submit` method:

C#

```

Logger.LogInformation("Id = {Id} Description = {Description} " +
    "Classification = {Classification} MaximumAccommodation = " +
    "{MaximumAccommodation} IsValidatedDesign = " +
    "{IsValidatedDesign} ProductionDate = {ProductionDate} " +
    "Manufacturer = {Manufacturer}, Engine = {Engine}, " +
    "Color = {Color}",
    Model?.Id, Model?.Description, Model?.Classification,
    Model?.MaximumAccommodation, Model?.IsValidatedDesign,
    Model?.ProductionDate, Model?.Manufacturer, Model?.Engine,
    Model?.Color);

```

ASP.NET Core Blazor forms validation

Article • 10/18/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to use validation in Blazor forms.

Form validation

In basic form validation scenarios, an [EditForm](#) instance can use declared [EditContext](#) and [ValidationMessageStore](#) instances to validate form fields. A handler for the [OnValidationRequested](#) event of the [EditContext](#) executes custom validation logic. The handler's result updates the [ValidationMessageStore](#) instance.

Basic form validation is useful in cases where the form's model is defined within the component hosting the form, either as members directly on the component or in a subclass. Use of a [validator component](#) is recommended where an independent model class is used across several components.

In Blazor Web Apps, client-side validation requires an active Blazor SignalR circuit. Client-side validation isn't available to forms in components that have adopted static server-side rendering (static SSR). Forms that adopt static SSR are validated on the server after the form is submitted.

In the following component, the `HandleValidationRequested` handler method clears any existing validation messages by calling [ValidationMessageStore.Clear](#) before validating the form.

Starship8.razor:

```
razor
```

```
@page "/starship-8"  
@implements IDisposable  
@inject ILogger<Starship8> Logger
```

```
<h2>Holodeck Configuration</h2>
```

```
<EditForm EditContext="editContext" OnValidSubmit="Submit"
FormName="Starship8">
    <div>
        <label>
            <InputCheckbox @bind-Value="Model!.Subsystem1" />
            Safety Subsystem
        </label>
    </div>
    <div>
        <label>
            <InputCheckbox @bind-Value="Model!.Subsystem2" />
            Emergency Shutdown Subsystem
        </label>
    </div>
    <div>
        <ValidationMessage For="() => Model!.Options" />
    </div>
    <div>
        <button type="submit">Update</button>
    </div>
</EditForm>
```

```
@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Holodeck? Model { get; set; }

    private ValidationMessageStore? messageStore;

    protected override void OnInitialized()
    {
        Model ??= new();
        editContext = new(Model);
        editContext.OnValidationRequested += HandleValidationRequested;
        messageStore = new(editContext);
    }

    private void HandleValidationRequested(object? sender,
        ValidationRequestedEventArgs args)
    {
        messageStore?.Clear();

        // Custom validation logic
        if (!Model!.Options)
        {
            messageStore?.Add(() => Model.Options, "Select at least one.");
        }
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");
}
```

```



public class Holodeck
{
    public bool Subsystem1 { get; set; }
    public bool Subsystem2 { get; set; }
    public bool Options => Subsystem1 || Subsystem2;
}

public void Dispose()
{
    if (editContext is not null)
    {
        editContext.OnValidationRequested -= HandleValidationRequested;
    }
}
}


```

Data Annotations Validator component and custom validation

The [DataAnnotationsValidator](#) component attaches data annotations validation to a cascaded [EditContext](#). Enabling data annotations validation requires the [DataAnnotationsValidator](#) component. To use a different validation system than data annotations, use a custom implementation instead of the [DataAnnotationsValidator](#) component. The framework implementations for [DataAnnotationsValidator](#) are available for inspection in the reference source:

- [DataAnnotationsValidator](#) 
- [AddDataAnnotationsValidation](#)  .

Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#) .

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the [DataAnnotationsValidator](#) component associates all reported validation results with the field.

- *Model validation* is performed when the user submits the form. During model validation, the [DataAnnotationsValidator](#) component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validator components

Validator components support form validation by managing a [ValidationMessageStore](#) for a form's [EditContext](#).

The Blazor framework provides the [DataAnnotationsValidator](#) component to attach validation support to forms based on [validation attributes \(data annotations\)](#). You can create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing (for example, client validation followed by server validation). The validator component example shown in this section, `CustomValidation`, is used in the following sections of this article:

- [Business logic validation with a validator component](#)
- [Server validation with a validator component](#)

Of the [data annotation built-in validators](#), only the [\[Remote\] validation attribute](#) isn't supported in Blazor.

ⓘ Note

Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the [DataAnnotationsValidator](#) component. When used with server validation, any custom attributes applied to the model must be executable on the server. For more information, see the [Custom validation attributes](#) section.

Create a validator component from [ComponentBase](#):

- The form's [EditContext](#) is a [cascading parameter](#) of the component.
- When the validator component is initialized, a new [ValidationMessageStore](#) is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors`

method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.

- Messages are cleared when any of the following have occurred:
 - Validation is requested on the `EditContext` when the `OnValidationRequested` event is raised. All of the errors are cleared.
 - A field changes in the form when the `OnFieldChanged` event is raised. Only the errors for the field are cleared.
 - The `ClearErrors` method is called by developer code. All of the errors are cleared.

Update the namespace in the following class to match your app's namespace.

`CustomValidation.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample;

public class CustomValidation : ComponentBase
{
    private ValidationMessageStore? messageStore;

    [CascadingParameter]
    private EditContext? CurrentEditContext { get; set; }

    protected override void OnInitialized()
    {
        if (CurrentEditContext is null)
        {
            throw new InvalidOperationException(
                $"{nameof(CustomValidation)} requires a cascading " +
                $"parameter of type {nameof(EditContext)}. " +
                $"For example, you can use {nameof(CustomValidation)} " +
                $"inside an {nameof(EditForm)}." );
        }

        messageStore = new(CurrentEditContext);

        CurrentEditContext.OnValidationRequested += (s, e) =>
            messageStore?.Clear();
        CurrentEditContext.OnFieldChanged += (s, e) =>
            messageStore?.Clear(e.FieldIdentifier);
    }

    public void DisplayErrors(Dictionary<string, List<string>> errors)
    {
        if (CurrentEditContext is not null)
        {

```

```

        foreach (var err in errors)
        {
            messageStore?.Add(CurrentEditContext.Field(err.Key),
err.Value);
        }

        CurrentEditContext.NotifyValidationStateChanged();
    }

    public void ClearErrors()
    {
        messageStore?.Clear();
        CurrentEditContext?.NotifyValidationStateChanged();
    }
}

```

Important

Specifying a namespace is **required** when deriving from [ComponentBase](#). Failing to specify a namespace results in a build error:

Tag helpers cannot target tag name '<global namespace>.{CLASS NAME}' because it contains a ' ' character.

The `{CLASS NAME}` placeholder is the name of the component class. The custom validator example in this section specifies the example namespace `BlazorSample`.

Note

Anonymous lambda expressions are registered event handlers for [OnValidationRequested](#) and [OnFieldChanged](#) in the preceding example. It isn't necessary to implement [IDisposable](#) and unsubscribe the event delegates in this scenario. For more information, see [ASP.NET Core Razor component lifecycle](#).

Business logic validation with a validator component

For general business logic validation, use a [validator component](#) that receives form errors in a dictionary.

Basic validation is useful in cases where the form's model is defined within the component hosting the form, either as members directly on the component or in a subclass. Use of a validator component is recommended where an independent model class is used across several components.

In the following example:

- A shortened version of the `Starfleet Starship Database` form (`Starship3` component) of the [Example form](#) section of the *Input components* article is used that only accepts the starship's classification and description. Data annotation validation isn't triggered on form submission because the [DataAnnotationsValidator](#) component isn't included in the form.
- The `CustomValidation` component from the [Validator components](#) section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the "Defense" ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's [ValidationMessageStore](#) and shown in the [EditForm](#)'s validation summary.

`Starship9.razor`:

razor

```
@page "/starship-9"
@inject ILogger<Starship9> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship9">
    <CustomValidation @ref="customValidation" />
    <ValidationSummary />
    <div>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="Model!.Classification">
                <option value="">
                    Select classification ...
                </option>
                <option checked="@ (Model!.Classification == "Exploration")"
                    value="Exploration">
                    Exploration
                </option>
                <option checked="@ (Model!.Classification == "Diplomacy")"
                    value="Diplomacy">
                    Diplomacy
                </option>
            </InputSelect>
        </label>
    </div>
</EditForm>
```



```

        <option checked="@ (Model!.Classification == "Defense")"
            value="Defense">
            Defense
        </option>
    </InputSelect>
</label>
</div>
<div>
    <label>
        Description (optional):
        <InputTextArea @bind-Value="Model!.Description" />
    </label>
</div>
<div>
    <button type="submit">Submit</button>
</div>
</EditForm>

@code {
    private CustomValidation? customValidation;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() =>
        Model ??= new() { ProductionDate = DateTime.UtcNow };

    private void Submit()
    {
        customValidation?.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (Model!.Classification == "Defense" &&
            string.IsNullOrEmpty(Model.Description))
        {
            errors.Add(nameof(Model.Description),
                new() { "For a 'Defense' ship classification, " +
                    "'Description' is required." });
        }

        if (errors.Any())
        {
            customValidation?.DisplayErrors(errors);
        }
        else
        {
            Logger.LogInformation("Submit called: Processing the form");
        }
    }
}

```

As an alternative to using [validation components](#), data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the [DataAnnotationsValidator](#) component. When used with server validation, the attributes must be executable on the server. For more information, see the [Custom validation attributes](#) section.

Server validation with a validator component

This section is focused on Blazor Web App scenarios, but the approach for any type of app that uses server validation with web API adopts the same general approach.

Server validation is supported in addition to client validation:

- Process client validation in the form with the [DataAnnotationsValidator](#) component.
- When the form passes client validation ([OnValidSubmit](#) is called), send the [EditContext.Model](#) to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code ([200 - OK](#)). If validation fails, return a failure status code ([400 - Bad Request](#)) and the field validation errors.
- Either disable the form on success or display the errors.

Basic validation is useful in cases where the form's model is defined within the component hosting the form, either as members directly on the component or in a subclass. Use of a validator component is recommended where an independent model class is used across several components.

The following example is based on:

- A Blazor Web App with Interactive WebAssembly components created from the [Blazor Web App project template](#).
- The `Starship` model (`Starship.cs`) of the [Example form](#) section of the *Input components* article.
- The `CustomValidation` component shown in the [Validator components](#) section.

Place the `Starship` model (`Starship.cs`) into a shared class library project so that both the client and server projects can use the model. Add or update the namespace to match the namespace of the shared app (for example, `namespace BlazorSample.Shared`).

Since the model requires data annotations, confirm that the shared class library uses the shared framework or add the [System.ComponentModel.Annotations](#) package to the shared project.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

In the main project of the Blazor Web App, add a controller to process starship validation requests and return failed validation messages. Update the namespaces in the last `using` statement for the shared class library project and the `namespace` for the controller class. In addition to client and server data annotations validation, the controller validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

The validation for the `Defense` ship classification only occurs on the server in the controller because the upcoming form doesn't perform the same validation client-side when the form is submitted to the server. Server validation without client validation is common in apps that require private business logic validation of user input on the server. For example, private information from data stored for a user might be required to validate user input. Private data obviously can't be sent to the client for client validation.

ⓘ Note

The `StarshipValidation` controller in this section uses Microsoft Identity 2.0. The Web API only accepts tokens for users that have the "`API.Access`" scope for this API. Additional customization is required if the API's scope name is different from `API.Access`.

For more information on security, see:

- [ASP.NET Core Blazor authentication and authorization](#) (and the other articles in the *Blazor Security and Identity* node)
- [Microsoft identity platform documentation](#)

`Controllers/StarshipValidation.cs`:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers;

[Authorize]
[ApiController]
[Route("[controller]")]
public class StarshipValidationController(
    ILogger<StarshipValidationController> logger)
    : ControllerBase
{
    static readonly string[] scopeRequiredByApi = new[] { "API.Access" };

    [HttpPost]
    public async Task<IActionResult> Post(Starship model)
    {
        HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);

        try
        {
            if (model.Classification == "Defense" &&
                string.IsNullOrEmpty(model.Description))
            {
                ModelState.AddModelError(nameof(model.Description),
                    "For a 'Defense' ship " +
                    "classification, 'Description' is required.");
            }
            else
            {
                logger.LogInformation("Processing the form asynchronously");

                // async ...

                return Ok(ModelState);
            }
        }
        catch (Exception ex)
        {
            logger.LogError("Validation Error: {Message}", ex.Message);
        }

        return BadRequest(ModelState);
    }
}
```

Confirm or update the namespace of the preceding controller
(`BlazorSample.Server.Controllers`) to match the app's controllers' namespace.

When a model binding validation error occurs on the server, an [ApiController](#) ([ApiControllerAttribute](#)) normally returns a [default bad request response](#) with a [ValidationProblemDetails](#). The response contains more data than just the validation errors, as shown in the following example when all of the fields of the `Starfleet Starship Database` form aren't submitted and the form fails validation:

JSON

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Id": ["The Id field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

❗ Note

To demonstrate the preceding JSON response, you must either disable the form's client validation to permit empty field form submission or use a tool to send a request directly to the server API, such as [Firefox Browser Developer](#) [↗].

If the server API returns the preceding default JSON response, it's possible for the client to parse the response in developer code to obtain the children of the `errors` node for forms validation error processing. It's inconvenient to write developer code to parse the file. Parsing the JSON manually requires producing a [Dictionary<string, List<string>>](#) of errors after calling [ReadFromJsonAsync](#). Ideally, the server API should only return the validation errors, as the following example shows:

JSON

```
{
  "Id": ["The Id field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with [ApiControllerAttribute](#) in the `Program` file. For the API endpoint (`/StarshipValidation`), return a

[BadRequestObjectResult](#) with the [ModelStateDictionary](#). For any other API endpoints, preserve the default behavior by returning the object result with a new [ValidationProblemDetails](#).

Add the [Microsoft.AspNetCore.Mvc](#) namespace to the top of the `Program` file in the main project of the Blazor Web App:

C#

```
using Microsoft.AspNetCore.Mvc;
```

In the `Program` file, add or update the following [AddControllersWithViews](#) extension method and add the following call to [ConfigureApiBehaviorOptions](#):

C#

```
builder.Services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

If you're adding controllers to the main project of the Blazor Web App for the first time, map controller endpoints when you place the preceding code that registers services for controllers. The following example uses default controller routes:

C#

```
app.MapDefaultControllerRoute();
```

ⓘ Note

The preceding example explicitly registers controller services by calling [AddControllersWithViews](#) to automatically [mitigate Cross-Site Request Forgery](#).

(XSRF/CSRF) attacks. If you merely use AddControllers, antiforgery isn't enabled automatically.

For more information on controller routing and validation failure error responses, see the following resources:

- [Routing to controller actions in ASP.NET Core](#)
- [Handle errors in ASP.NET Core controller-based web APIs](#)

In the `.Client` project, add the `CustomValidation` component shown in the [Validator components](#) section. Update the namespace to match the app (for example, `namespace BlazorSample.Client`).

In the `.Client` project, the `Starfleet Starship Database` form is updated to show server validation errors with help of the `CustomValidation` component. When the server API returns validation messages, they're added to the `CustomValidation` component's [ValidationMessageStore](#). The errors are available in the form's [EditContext](#) for display by the form's validation summary.

In the following component, update the namespace of the shared project (`@using BlazorSample.Shared`) to the shared project's namespace. Note that the form requires authorization, so the user must be signed into the app to navigate to the form.

`Starship10.razor`:

ⓘ Note

Forms based on [EditForm](#) automatically enable [antiforgery support](#). The controller should use [AddControllersWithViews](#) to register controller services and automatically enable antiforgery support for the web API.

razor

```
@page "/starship-10"
@rendermode InteractiveWebAssembly
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<Starship10> Logger
```

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

```
<EditForm FormName="Starship10" Model="Model" OnValidSubmit="Submit">
  <DataAnnotationsValidator />
  <CustomValidation @ref="customValidation" />
  <ValidationSummary />
  <div>
    <label>
      Identifier:
      <InputText @bind-Value="Model!.Id" disabled="@disabled" />
    </label>
  </div>
  <div>
    <label>
      Description (optional):
      <InputTextArea @bind-Value="Model!.Description"
        disabled="@disabled" />
    </label>
  </div>
  <div>
    <label>
      Primary Classification:
      <InputSelect @bind-Value="Model!.Classification"
disabled="@disabled">
        <option value="">Select classification ...</option>
        <option value="Exploration">Exploration</option>
        <option value="Diplomacy">Diplomacy</option>
        <option value="Defense">Defense</option>
      </InputSelect>
    </label>
  </div>
  <div>
    <label>
      Maximum Accommodation:
      <InputNumber @bind-Value="Model!.MaximumAccommodation"
        disabled="@disabled" />
    </label>
  </div>
  <div>
    <label>
      Engineering Approval:
      <InputCheckbox @bind-Value="Model!.IsValidatedDesign"
        disabled="@disabled" />
    </label>
  </div>
  <div>
    <label>
      Production Date:
      <InputDate @bind-Value="Model!.ProductionDate"
disabled="@disabled" />
    </label>
  </div>
</div>
```



```

        <button type="submit" disabled="@disabled">Submit</button>
    </div>
    <div style="@messageStyles">
        @message
    </div>
</EditForm>

@code {
    private CustomValidation? customValidation;
    private bool disabled;
    private string? message;
    private string messageStyles = "visibility:hidden";

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() =>
        Model ??= new() { ProductionDate = DateTime.UtcNow };

    private async Task Submit(EditContext editContext)
    {
        customValidation?.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>() ??
                new Dictionary<string, List<string>>());

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Any())
            {
                customValidation?.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code:
{response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)

```

```

    {
        Logger.LogError("Form processing error: {Message}", ex.Message);
        disabled = true;
        messageStyles = "color:red";
        message = "There was an error processing the form.";
    }
}
}

```

The `.Client` project of a Blazor Web App must also register an `HttpClient` for HTTP POST requests to a backend web API controller. Confirm or add the following to the `.Client` project's `Program` file:

```

C#

builder.Services.AddScoped(sp =>
    new HttpClient { BaseAddress = new
    Uri(builder.HostEnvironment.BaseAddress) });

```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (`IWebAssemblyHostEnvironment.BaseAddress`), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page.

ⓘ Note

As an alternative to the use of a [validation component](#), data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the [DataAnnotationsValidator](#) component. When used with server validation, the attributes must be executable on the server. For more information, see the [Custom validation attributes](#) section.

InputText based on the input event

Use the `InputText` component to create a custom component that uses the `oninput` event ([input](#)) instead of the `onchange` event ([change](#)). Use of the `input` event triggers field validation on each keystroke.

The following `CustomInputText` component inherits the framework's `InputText` component and sets event binding to the `oninput` event ([input](#)).

`CustomInputText.razor`:

razor

```
@inherits InputText

<input @attributes="AdditionalAttributes"
      class="@CssClass"
      @bind="CurrentValueAsString"
      @bind:event="oninput" />
```

The `CustomInputText` component can be used anywhere `InputText` is used. The following component uses the shared `CustomInputText` component.

Starship11.razor:

razor

```
@page "/starship-11"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship11> Logger

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship11">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <CustomInputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

<div>
    CurrentValue: @Model?.Id
</div>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Submit: Processing form");

    public class Starship
    {
        [Required]
        [StringLength(10, ErrorMessage = "Id is too long.")]
        public string? Id { get; set; }
    }
}
```

```
}  
}
```

Validation Summary and Validation Message components

The [ValidationSummary](#) component summarizes all validation messages, which is similar to the [Validation Summary Tag Helper](#):

```
razor
```

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
razor
```

```
<ValidationSummary Model="Model" />
```

The [ValidationMessage<TValue>](#) component displays validation messages for a specific field, which is similar to the [Validation Message Tag Helper](#). Specify the field for validation with the [For](#) attribute and a lambda expression naming the model property:

```
razor
```

```
<ValidationMessage For="@(() => Model!.MaximumAccommodation)" />
```

The [ValidationMessage<TValue>](#) and [ValidationSummary](#) components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
css
```

```
.validation-message {  
    color: red;  
}
```

Determine if a form field is valid

Use [EditContext.IsValid](#) to determine if a field is valid without obtaining validation messages.

✗ Supported, but not recommended:

C#

```
var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();
```

✓ Recommended:

C#

```
var isValid = editContext.IsValid(fieldIdentifier);
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a [custom validation attribute](#), pass the validation context's [MemberName](#) when creating the [ValidationResult](#).

CustomValidator.cs:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

public class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

Inject services into custom validation attributes through the [ValidationContext](#). The following example demonstrates a salad chef form that validates user input with dependency injection (DI).

The `SaladChef` class indicates the approved starship ingredient list for a Ten Forward salad.

`SaladChef.cs`:

C#

```
namespace BlazorSample;

public class SaladChef
{
    public string[] SaladToppers = { "Horva", "Kanda Root", "Krintar",
    "Plomeek",
    "Syto Bean" };
}
```

Register `SaladChef` in the app's DI container in the `Program` file:

C#

```
builder.Services.AddTransient<SaladChef>();
```

The `IsValid` method of the following `SaladChefValidatorAttribute` class obtains the `SaladChef` service from DI to check the user's input.

`SaladChefValidatorAttribute.cs`:

C#

```
using System.ComponentModel.DataAnnotations;

namespace BlazorSample;

public class SaladChefValidatorAttribute : ValidationAttribute
{
    protected override ValidationResult? IsValid(object? value,
        ValidationContext validationContext)
    {
        var saladChef = validationContext.GetRequiredService<SaladChef>();

        if (saladChef.SaladToppers.Contains(value?.ToString()))
        {
            return ValidationResult.Success;
        }

        return new ValidationResult("Is that a Vulcan salad topper?! " +
            "The following toppers are available for a Ten Forward salad: "
            +
            string.Join(", ", saladChef.SaladToppers));
    }
}
```

```
}  
}
```

The following component validates user input by applying the `SaladChefValidatorAttribute` (`[SaladChefValidator]`) to the salad ingredient string (`SaladIngredient`).

`Starship12.razor`:

razor

```
@page "/starship-12"  
@inject SaladChef SaladChef  
  
<EditForm Model="this" autocomplete="off" FormName="Starship12">  
    <DataAnnotationsValidator />  
    <div>  
        <label>  
            Salad topper (@saladToppers):  
            <input @bind="SaladIngredient" />  
        </label>  
    </div>  
    <div>  
        <button type="submit">Submit</button>  
    </div>  
    <ul>  
        @foreach (var message in context.GetValidationMessages())  
        {  
            <li class="validation-message">@message</li>  
        }  
    </ul>  
</EditForm>  
  
@code {  
    private string? saladToppers;  
  
    [SaladChefValidator]  
    public string? SaladIngredient { get; set; }  
  
    protected override void OnInitialized() =>  
        saladToppers ??= string.Join(", ", SaladChef.SaladToppers);  
}
```

Custom validation CSS class attributes

Custom validation CSS class attributes are useful when integrating with CSS frameworks, such as [Bootstrap](#).

To specify custom validation CSS class attributes, start by providing CSS styles for custom validation. In the following example, valid (`validField`) and invalid (`invalidField`) styles are specified.

Add the following CSS classes to the app's stylesheet:

```
CSS

.validField {
    border-color: lawngreen;
}

.invalidField {
    background-color: tomato;
}
```

Create a class derived from `FieldCssClassProvider` that checks for field validation messages and applies the appropriate valid or invalid style.

`CustomFieldClassProvider.cs`:

```
C#

using Microsoft.AspNetCore.Components.Forms;

public class CustomFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = editContext.IsValid(fieldIdentifier);

        return isValid ? "validField" : "invalidField";
    }
}
```

Set the `CustomFieldClassProvider` class as the Field CSS Class Provider on the form's `EditContext` instance with `SetFieldCssClassProvider`.

`Starship13.razor`:

```
razor

@page "/starship-13"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship13> Logger

<EditForm EditContext="editContext" OnValidSubmit="Submit">
```



```

FormName="Starship13">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??= new();
        editContext = new(Model);
        editContext.SetFieldCssClassProvider(new
CustomFieldClassProvider());
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    public class Starship
    {
        [Required]
        [StringLength(10, ErrorMessage = "Id is too long.")]
        public string? Id { get; set; }
    }
}

```

The preceding example checks the validity of all form fields and applies a style to each field. If the form should only apply custom styles to a subset of the fields, make `CustomFieldClassProvider` apply styles conditionally. The following `CustomFieldClassProvider2` example only applies a style to the `Name` field. For any fields with names not matching `Name`, `string.Empty` is returned, and no style is applied. Using [reflection](#), the field is matched to the model member's property or field name, not an `id` assigned to the HTML entity.

`CustomFieldClassProvider2.cs`:

```
using Microsoft.AspNetCore.Components.Forms;

public class CustomFieldClassProvider2 : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        if (fieldIdentifier.FieldName == "Name")
        {
            var isValid = editContext.IsValid(fieldIdentifier);

            return isValid ? "validField" : "invalidField";
        }

        return string.Empty;
    }
}
```

ⓘ Note

Matching the field name in the preceding example is case sensitive, so a model property member designated "Name" must match a conditional check on "Name":

- ✔ fieldId.FieldName == "Name"
- ✗ fieldId.FieldName == "name"
- ✗ fieldId.FieldName == "NAME"
- ✗ fieldId.FieldName == "nAmE"

Add an additional property to `Model`, for example:

C#

```
[StringLength(10, ErrorMessage = "Description is too long.")]
public string? Description { get; set; }
```

Add the `Description` to the `CustomValidationForm` component's form:

razor

```
<InputText @bind-Value="Model!.Description" />
```

Update the `EditContext` instance in the component's `OnInitialized` method to use the new Field CSS Class Provider:

C#

```
editContext?.SetFieldCssClassProvider(new CustomFieldClassProvider2());
```

Because a CSS validation class isn't applied to the `Description` field, it isn't styled. However, field validation runs normally. If more than 10 characters are provided, the validation summary indicates the error:

Description is too long.

In the following example:

- The custom CSS style is applied to the `Name` field.
- Any other fields apply logic similar to Blazor's default logic and using Blazor's default field CSS validation styles, modified with `valid` or `invalid`. Note that for the default styles, you don't need to add them to the app's stylesheet if the app is based on a Blazor project template. For apps not based on a Blazor project template, the default styles can be added to the app's stylesheet:

CSS

```
.valid.modified:not([type=checkbox]) {  
    outline: 1px solid #26b050;  
}  
  
.invalid {  
    outline: 1px solid red;  
}
```

CustomFieldClassProvider3.cs:

C#

```
using Microsoft.AspNetCore.Components.Forms;  
  
public class CustomFieldClassProvider3 : FieldCssClassProvider  
{  
    public override string GetFieldCssClass(EditContext editContext,  
        in FieldIdentifier fieldIdentifier)  
    {  
        var isValid = editContext.IsValid(fieldIdentifier);  
  
        if (fieldIdentifier.FieldName == "Name")  
        {  
            return isValid ? "validField" : "invalidField";  
        }  
    }  
}
```

```

else
{
    if (editContext.IsModified(fieldIdentifier))
    {
        return isValid ? "modified valid" : "modified invalid";
    }
    else
    {
        return isValid ? "valid" : "invalid";
    }
}
}
}

```

Update the `EditContext` instance in the component's `OnInitialized` method to use the preceding Field CSS Class Provider:

C#

```
editContext.SetFieldCssClassProvider(new CustomFieldClassProvider3());
```

Using `CustomFieldClassProvider3`:

- The `Name` field uses the app's custom validation CSS styles.
- The `Description` field uses logic similar to Blazor's logic and Blazor's default field CSS validation styles.

Class-level validation with `IValidatableObject`

[Class-level validation with IValidatableObject \(API documentation\)](#) is supported for Blazor form models. `IValidatableObject` validation only executes when the form is submitted and only if all other validation succeeds.

Blazor data annotations validation package

The [Microsoft.AspNetCore.Components.DataAnnotations.Validation](#) [↗] is a package that fills validation experience gaps using the `DataAnnotationsValidator` component. The package is currently *experimental*.

Warning

The [Microsoft.AspNetCore.Components.DataAnnotations.Validation](#) [↗] package has a latest version of *release candidate* at [NuGet.org](#) [↗]. Continue to use the

experimental release candidate package at this time. Experimental features are provided for the purpose of exploring feature viability and may not ship in a stable version. Watch the [Announcements GitHub repository](#), the [dotnet/aspnetcore GitHub repository](#), or this topic section for further updates.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in [DataAnnotationsValidator](#). However, the [DataAnnotationsValidator](#) only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* [Microsoft.AspNetCore.Components.DataAnnotations.Validation](#) package:

razor

```
<EditForm ...>
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

Starship.cs:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } = new();

    ...
}
```

ShipDescription.cs:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string? ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string? LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation, the following example:

- Uses a shortened version of the earlier `Starfleet Starship Database` form (`Starship3` component) of the [Example form](#) section of the *Input components* article that only accepts a value for the ship's Id. The other `Starship` properties receive valid default values when an instance of the `Starship` type is created.
- Uses the form's `EditContext` to assign the model when the component is initialized.
- Validates the form in the context's `OnFieldChanged` callback to enable and disable the submit button.
- Implements `IDisposable` and unsubscribes the event handler in the `Dispose` method. For more information, see [ASP.NET Core Razor component lifecycle](#).

ⓘ Note

When assigning to the `EditForm.EditContext`, don't also assign an `EditForm.Model` to the `EditForm`.

Starship14.razor:

razor

```

@page "/starship-14"
@implements IDisposable
@Inject ILogger<Starship14> Logger

<EditForm EditContext="editContext" OnValidSubmit="Submit"
FormName="Starship14">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit" disabled="@formInvalid">Submit</button>
    </div>
</EditForm>

@code {
    private bool formInvalid = false;
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??=
            new()
            {
                Id = "NCC-1701",
                Classification = "Exploration",
                MaximumAccommodation = 150,
                IsValidatedDesign = true,
                ProductionDate = new DateTime(2245, 4, 11)
            };
        editContext = new(Model);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object? sender, FieldChangedEventArgs e)
    {
        if (editContext is not null)
        {
            formInvalid = !editContext.Validate();
            StateHasChanged();
        }
    }

    private void Submit() => Logger.LogInformation("Submit: Processing form");

    public void Dispose()

```

```

    {
        if (editContext is not null)
        {
            editContext.OnFieldChanged -= HandleFieldChanged;
        }
    }
}

```

If a form isn't preloaded with valid values and you wish to disable the **Submit** button on form load, set `formInvalid` to `true`.

A side effect of the preceding approach is that a validation summary ([ValidationSummary](#) component) is populated with invalid fields after the user interacts with any one field. Address this scenario in either of the following ways:

- Don't use a [ValidationSummary](#) component on the form.
- Make the [ValidationSummary](#) component visible when the submit button is selected (for example, in a `Submit` method).

razor

```

<EditForm ... EditContext="editContext" OnValidSubmit="Submit" ...>
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void Submit()
    {
        displaySummary = "display:block";
    }
}

```


Troubleshoot ASP.NET Core Blazor forms

Article • 04/11/2024

📘 Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article provides troubleshooting guidance for Blazor forms.

Large form payloads and the SignalR message size limit

This section only applies to Blazor Web Apps, Blazor Server apps, and hosted Blazor WebAssembly solutions that implement SignalR.

If form processing fails because the component's form payload has exceeded the maximum incoming SignalR message size permitted for hub methods, the form can adopt [streaming JS interop](#) without increasing the message size limit. For more information on the size limit and the error thrown, see [ASP.NET Core Blazor SignalR guidance](#).

In the following example a text area (`<textarea>`) is used with streaming JS interop to move up to 50,000 bytes of data to the server.

Add a JavaScript (JS) `getText` function to the app:

JavaScript

```
window.getText = (elem) => {  
    const textValue = elem.value;  
    const utf8Encoder = new TextEncoder();  
    const encodedTextValue = utf8Encoder.encode(textValue);  
    return encodedTextValue;  
};
```

For information on where to place JS in a Blazor app, see [JavaScript location in ASP.NET Core Blazor apps](#).

Due to security considerations, zero-length streams aren't permitted for streaming JS Interop. Therefore, the following `StreamFormData` component traps a [JSException](#) and returns an empty string if the text area is blank when the form is submitted.

`StreamFormData.razor`:

razor

```
@page "/stream-form-data"
@inject IJSRuntime JS
@inject ILogger<StreamFormData> Logger

<h1>Stream form data with JS interop</h1>

<EditForm FormName="StreamFormData" Model="this" OnSubmit="Submit">
    <div>
        <label>
            &lt;textarea&gt; value streamed for assignment to
            <code>TextAreaValue (&lt;= 50,000 characters)</code>:
            <textarea @ref="largeTextArea" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

<div>
    Length: @TextAreaValue?.Length
</div>

@code {
    private ElementReference largeTextArea;

    public string? TextAreaValue { get; set; }

    protected override void OnInitialized() =>
        TextAreaValue ??= string.Empty;

    private async Task Submit()
    {
        TextAreaValue = await GetTextAsync();

        Logger.LogInformation("TextAreaValue length: {Length}",
            TextAreaValue.Length);
    }

    public async Task<string> GetTextAsync()
    {
        try
        {
            var streamRef =
                await JS.InvokeAsync<IJSStreamReference>("getText",
```

```

largeTextArea);
        var stream = await streamRef.OpenReadStreamAsync(maxAllowedSize:
50_000);
        var streamReader = new StreamReader(stream);

        return await streamReader.ReadToEndAsync();
    }
    catch (JSEException jsException)
    {
        if (jsException.InnerException is
            ArgumentOutOfRangeException outOfRangeException &&
            outOfRangeException.ActualValue is not null &&
            outOfRangeException.ActualValue is long actualLength &&
            actualLength == 0)
        {
            return string.Empty;
        }

        throw;
    }
}
}
}

```

EditForm parameter error

InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the [EditForm](#) assigns a [Model](#) or an [EditContext](#). Don't use both for the same form.

When assigning to [Model](#), confirm that the model type is instantiated.

Connection disconnected

Error: Connection disconnected with error 'Error: Server returned an error on close: Connection closed with an error.'

System.IO.InvalidDataException: The maximum message size of 32768B was exceeded. The message size can be configured in AddHubOptions.

For more information and guidance, see the following resources:

- [Large form payloads and the SignalR message size limit](#)
- [ASP.NET Core Blazor SignalR guidance](#)

ASP.NET Core Blazor file uploads

Article • 10/04/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to upload files in Blazor with the `InputFile` component.

File uploads

Warning

Always follow security best practices when permitting users to upload files. For more information, see [Upload files in ASP.NET Core](#).

Use the `InputFile` component to read browser file data into .NET code. The `InputFile` component renders an HTML `<input>` element of type `file` for single file uploads. Add the `multiple` attribute to permit the user to upload multiple files at once.

File selection isn't cumulative when using an `InputFile` component or its underlying HTML `<input type="file">` [↗](#), so you can't add files to an existing file selection. The component always replaces the user's initial file selection, so file references from prior selections aren't available.

The following `InputFile` component executes the `LoadFiles` method when the `OnChange` (`change` [↗](#)) event occurs. An `InputFileChangeEventArgs` provides access to the selected file list and details about each file:

razor

```
<InputFile OnChange="LoadFiles" multiple />

@code {
    private void LoadFiles(InputFileChangeEventArgs e)
    {
        ...
    }
}
```

```
}  
}
```

Rendered HTML:

HTML

```
<input multiple="" type="file" _bl_2="">
```

ⓘ Note

In the preceding example, the `<input>` element's `_bl_2` attribute is used for Blazor's internal processing.

To read data from a user-selected file, call [IBrowserFile.OpenReadStream](#) on the file and read from the returned stream. For more information, see the [File streams](#) section.

[OpenReadStream](#) enforces a maximum size in bytes of its [Stream](#). Reading one file or multiple files larger than 500 KB results in an exception. This limit prevents developers from accidentally reading large files into memory. The `maxAllowedSize` parameter of [OpenReadStream](#) can be used to specify a larger size if required.

If you need access to a [Stream](#) that represents the file's bytes, use [IBrowserFile.OpenReadStream](#). Avoid reading the incoming file stream directly into memory all at once. For example, don't copy all of the file's bytes into a [MemoryStream](#) or read the entire stream into a byte array all at once. These approaches can result in degraded app performance and potential [Denial of Service \(DoS\)](#) risk, especially for server-side components. Instead, consider adopting either of the following approaches:

- Copy the stream directly to a file on disk without reading it into memory. Note that Blazor apps executing code on the server aren't able to access the client's file system directly.
- Upload files from the client directly to an external service. For more information, see the [Upload files to an external service](#) section.

In the following examples, `browserFile` represents the uploaded file and implements [IBrowserFile](#). Working implementations for [IBrowserFile](#) are shown in the file upload components later in this article.

✓ The following approach is **recommended** because the file's [Stream](#) is provided directly to the consumer, a [FileStream](#) that creates the file at the provided path:

C#

```
await using FileStream fs = new(path, FileMode.Create);
await browserFile.OpenReadStream().CopyToAsync(fs);
```

✓ The following approach is **recommended** for [Microsoft Azure Blob Storage](#) because the file's [Stream](#) is provided directly to [UploadBlobAsync](#):

C#

```
await blobContainerClient.UploadBlobAsync(
    trustedFileName, browserFile.OpenReadStream());
```

✗ The following approach is **NOT recommended** because the file's [Stream](#) content is read into a [String](#) in memory (`reader`):

C#

```
var reader =
    await new StreamReader(browserFile.OpenReadStream()).ReadToEndAsync();
```

✗ The following approach is **NOT recommended** for [Microsoft Azure Blob Storage](#) because the file's [Stream](#) content is copied into a [MemoryStream](#) in memory (`memoryStream`) before calling [UploadBlobAsync](#):

C#

```
var memoryStream = new MemoryStream();
await browserFile.OpenReadStream().CopyToAsync(memoryStream);
await blobContainerClient.UploadBlobAsync(
    trustedFileName, memoryStream);
```

A component that receives an image file can call the [BrowserFileExtensions.RequestImageFileAsync](#) convenience method on the file to resize the image data within the browser's JavaScript runtime before the image is streamed into the app. Use cases for calling [RequestImageFileAsync](#) are most appropriate for Blazor WebAssembly apps.

File size read and upload limits

Server-side or client-side, there's no file read or upload size limit specifically for the [InputFile](#) component. However, client-side Blazor reads the file's bytes into a single JavaScript array buffer when marshalling the data from JavaScript to C#, which is limited

to 2 GB or to the device's available memory. Large file uploads (> 250 MB) may fail for client-side uploads using the [InputFile](#) component. For more information, see the following discussions:

- [The Blazor InputFile Component should handle chunking when the file is uploaded \(dotnet/runtime #84685\)](#) [↗](#)
- [Request Streaming upload via http handler \(dotnet/runtime #36634\)](#) [↗](#)

For large client-side file uploads that fail when attempting to use the [InputFile](#) component, we recommend chunking large files with a custom component using multiple [HTTP range requests](#) [↗](#) instead of using the [InputFile](#) component.

Work is currently scheduled for .NET 9 (late 2024) to address the client-side file size upload limitation.

Examples

The following examples demonstrate multiple file upload in a component.

[InputFileChangeEventArgs.GetMultipleFiles](#) allows reading multiple files. Specify the maximum number of files to prevent a malicious user from uploading a larger number of files than the app expects. [InputFileChangeEventArgs.File](#) allows reading the first and only file if the file upload doesn't support multiple files.

[InputFileChangeEventArgs](#) is in the [Microsoft.AspNetCore.Components.Forms](#) namespace, which is typically one of the namespaces in the app's `_Imports.razor` file. When the namespace is present in the `_Imports.razor` file, it provides API member access to the app's components.

Namespaces in the `_Imports.razor` file aren't applied to C# files (`.cs`). C# files require an explicit [using](#) directive at the top of the class file:

razor

```
using Microsoft.AspNetCore.Components.Forms;
```

For testing file upload components, you can create test files of any size with [PowerShell](#):

PowerShell

```
$out = new-object byte[] {SIZE}; (new-object Random).NextBytes($out);  
[IO.File]::WriteAllBytes('{PATH}', $out)
```

In the preceding command:

- The `{SIZE}` placeholder is the size of the file in bytes (for example, `2097152` for a 2 MB file).
- The `{PATH}` placeholder is the path and file with file extension (for example, `D:/test_files/testfile2MB.txt`).

Server-side file upload example

To use the following code, create a `Development/unsafe_uploads` folder at the root of the app running in the `Development` environment.

Because the example uses the app's `environment` as part of the path where files are saved, additional folders are required if other environments are used in testing and production. For example, create a `Staging/unsafe_uploads` folder for the `Staging` environment. Create a `Production/unsafe_uploads` folder for the `Production` environment.

Warning

The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see [Upload files in ASP.NET Core](#).

`FileUpload1.razor`:

razor

```
@page "/file-upload-1"
@using System
@using System.IO
@using Microsoft.AspNetCore.Hosting
@inject ILogger<FileUpload1> Logger
@inject IWebHostEnvironment Environment

<PageTitle>File Upload 1</PageTitle>

<h1>File Upload Example 1</h1>

<p>
    <label>
        Max file size:
        <input type="number" @bind="maxFileSize" />
    </label>
</p>
```

```

<p>
    <label>
        Max allowed files:
        <input type="number" @bind="maxAllowedFiles" />
    </label>
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles of up to @maxFileSize bytes:
        <InputFile OnChange="LoadFiles" multiple />
    </label>
</p>

@if (isLoading)
{
    <p>Uploading...</p>
}
else
{
    <ul>
        @foreach (var file in loadedFiles)
        {
            <li>
                <ul>
                    <li>Name: @file.Name</li>
                    <li>Last modified: @file.LastModified.ToString()</li>
                    <li>Size (bytes): @file.Size</li>
                    <li>Content type: @file.ContentType</li>
                </ul>
            </li>
        }
    </ul>
}

@code {
    private List<IBrowserFile> loadedFiles = new();
    private long maxFileSize = 1024 * 15;
    private int maxAllowedFiles = 3;
    private bool isLoading;

    private async Task LoadFiles(InputFileChangeEventArgs e)
    {
        isLoading = true;
        loadedFiles.Clear();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            try
            {
                var trustedFileName = Path.GetRandomFileName();
                var path = Path.Combine(Environment.ContentRootPath,
                    Environment.EnvironmentName, "unsafe_uploads",
                    trustedFileName);
            }
        }
    }
}

```

```

        await using FileStream fs = new(path, FileMode.Create);
        await file.OpenReadStream(maxFileSize).CopyToAsync(fs);

        loadedFiles.Add(file);

        Logger.LogInformation(
            "Unsafe Filename: {UnsafeFilename} File saved:
{Filename}",
            file.Name, trustedFileName);
    }
    catch (Exception ex)
    {
        Logger.LogError("File: {Filename} Error: {Error}",
            file.Name, ex.Message);
    }
}

isLoading = false;
}
}

```

Client-side file upload example

The following example processes file bytes and doesn't send files to a destination outside of the app. For an example of a Razor component that sends a file to a server or service, see the following sections:

- [Upload files to a server with client-side rendering \(CSR\)](#)
- [Upload files to an external service](#)

The component assumes that the Interactive WebAssembly render mode (`InteractiveWebAssembly`) is inherited from a parent component or applied globally to the app.

razor

```

@page "/file-upload-1"
@inject ILogger<FileUpload1> Logger

<PageTitle>File Upload 1</PageTitle>

<h1>File Upload Example 1</h1>

<p>
    <label>
        Max file size:
        <input type="number" @bind="maxFileSize" />
    </label>
</p>

```

```

<p>
    <label>
        Max allowed files:
        <input type="number" @bind="maxAllowedFiles" />
    </label>
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles of up to @maxFileSize bytes:
        <InputFile OnChange="LoadFiles" multiple />
    </label>
</p>

@if (isLoading)
{
    <p>Uploading...</p>
}
else
{
    <ul>
        @foreach (var file in loadedFiles)
        {
            <li>
                <ul>
                    <li>Name: @file.Name</li>
                    <li>Last modified: @file.LastModified.ToString()</li>
                    <li>Size (bytes): @file.Size</li>
                    <li>Content type: @file.ContentType</li>
                </ul>
            </li>
        }
    </ul>
}

@code {
    private List<IBrowserFile> loadedFiles = new();
    private long maxFileSize = 1024 * 15;
    private int maxAllowedFiles = 3;
    private bool isLoading;

    private void LoadFiles(InputFileChangeEventArgs e)
    {
        isLoading = true;
        loadedFiles.Clear();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            try
            {
                loadedFiles.Add(file);
            }
            catch (Exception ex)
            {

```

```

        Logger.LogError("File: {FileName} Error: {Error}",
            file.Name, ex.Message);
    }
}

isLoading = false;
}
}

```

[IBrowserFile](#) returns metadata [exposed by the browser](#) as properties. Use this metadata for preliminary validation.

- [Name](#)
- [Size](#)
- [LastModified](#)
- [ContentType](#)

Never trust the values of the preceding properties, especially the [Name](#) property for display in the UI. Treat all user-supplied data as a significant security risk to the app, server, and network. For more information, see [Upload files in ASP.NET Core](#).

Upload files to a server with server-side rendering

This section applies to Interactive Server components in Blazor Web Apps or Blazor Server apps.

The following example demonstrates uploading files from a server-side app to a backend web API controller in a separate app, possibly on a separate server.

In the server-side app's `Program` file, add [IHttpClientFactory](#) and related services that allow the app to create [HttpClient](#) instances:

```

C#

builder.Services.AddHttpClient();

```

For more information, see [Make HTTP requests using IHttpClientFactory in ASP.NET Core](#).

For the examples in this section:

- The web API runs at the URL: `https://localhost:5001`
- The server-side app runs at the URL: `https://localhost:5003`

For testing, the preceding URLs are configured in the projects'

`Properties/launchSettings.json` files.

The following `UploadResult` class maintains the result of an uploaded file. When a file fails to upload on the server, an error code is returned in `ErrorCode` for display to the user. A safe file name is generated on the server for each file and returned to the client in `StoredFileName` for display. Files are keyed between the client and server using the unsafe/untrusted file name in `FileName`.

`UploadResult.cs`:

C#

```
public class UploadResult
{
    public bool Uploaded { get; set; }
    public string? FileName { get; set; }
    public string? StoredFileName { get; set; }
    public int ErrorCode { get; set; }
}
```

A security best practice for production apps is to avoid sending error messages to clients that might reveal sensitive information about an app, server, or network. Providing detailed error messages can aid a malicious user in devising attacks on an app, server, or network. The example code in this section only sends back an error code number (`int`) for display by the component client-side if a server-side error occurs. If a user requires assistance with a file upload, they provide the error code to support personnel for support ticket resolution without ever knowing the exact cause of the error.

The following `LazyBrowserFileStream` class defines a custom stream type that lazily calls `OpenReadStream` just before the first bytes of the stream are requested. The stream isn't transmitted from the browser to the server until reading the stream begins in .NET.

`LazyBrowserFileStream.cs`:

C#

```
using Microsoft.AspNetCore.Components.Forms;
using System.Diagnostics.CodeAnalysis;

namespace BlazorSample;

internal sealed class LazyBrowserFileStream(IBrowserFile file, int
maxAllowedSize)
    : Stream
```

```

{
    private readonly IBrowserFile file = file;
    private readonly int maxAllowedSize = maxAllowedSize;
    private Stream? underlyingStream;
    private bool isDisposed;

    public override bool CanRead => true;

    public override bool CanSeek => false;

    public override bool CanWrite => false;

    public override long Length => file.Size;

    public override long Position
    {
        get => underlyingStream?.Position ?? 0;
        set => throw new NotSupportedException();
    }

    public override void Flush() => underlyingStream?.Flush();

    public override Task<int> ReadAsync(byte[] buffer, int offset, int
count,
        CancellationToken cancellationToken)
    {
        EnsureStreamIsOpen();

        return underlyingStream.ReadAsync(buffer, offset, count,
cancellationToken);
    }

    public override ValueTask<int> ReadAsync(Memory<byte> buffer,
        CancellationToken cancellationToken = default)
    {
        EnsureStreamIsOpen();
        return underlyingStream.ReadAsync(buffer, cancellationToken);
    }

    [MemberNotNull(nameof(underlyingStream))]
    private void EnsureStreamIsOpen() =>
        underlyingStream ??= file.OpenReadStream(maxAllowedSize);

    protected override void Dispose(bool disposing)
    {
        if (isDisposed)
        {
            return;
        }

        underlyingStream?.Dispose();
        isDisposed = true;

        base.Dispose(disposing);
    }
}

```

```

    public override int Read(byte[] buffer, int offset, int count)
        => throw new NotSupportedException();

    public override long Seek(long offset, SeekOrigin origin)
        => throw new NotSupportedException();

    public override void SetLength(long value)
        => throw new NotSupportedException();

    public override void Write(byte[] buffer, int offset, int count)
        => throw new NotSupportedException();
}

```

The following `FileUpload2` component:

- Permits users to upload files from the client.
- Displays the untrusted/unsafe file name provided by the client in the UI. The untrusted/unsafe file name is automatically HTML-encoded by Razor for safe display in the UI.

Warning

Don't trust file names supplied by clients for:

- Saving the file to a file system or service.
- Display in UIs that don't encode file names automatically or via developer code.

For more information on security considerations when uploading files to a server, see [Upload files in ASP.NET Core](#).

`FileUpload2.razor`:

razor

```

@page "/file-upload-2"
@using System.Net.Http.Headers
@using System.Text.Json
@inject IHttpClientFactory ClientFactory
@inject ILogger<FileUpload2> Logger

<PageTitle>File Upload 2</PageTitle>

<h1>File Upload Example 2</h1>

<p>
    This example requires a backend server API to function. For more

```



```

information,
    see the <em>Upload files to a server</em> section
    of the <em>ASP.NET Core Blazor file uploads</em> article.
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles files:
        <InputFile OnChange="OnInputFileChange" multiple />
    </label>
</p>

@if (files.Any())
{
    <div class="card">
        <div class="card-body">
            <ul>
                @foreach (var file in files)
                {
                    <li>
                        File: @file.Name
                        <br>
                        @if (FileUpload(uploadResults, file.Name, Logger,
                            out var result))
                        {
                            <span>
                                Stored File Name: @result.StoredFileName
                            </span>
                        }
                        else
                        {
                            <span>
                                There was an error uploading the file
                                (Error: @result.ErrorCode).
                            </span>
                        }
                    </li>
                }
            </ul>
        </div>
    </div>
}

@code {
    private List<File> files = new();
    private List<UploadResult> uploadResults = new();
    private int maxAllowedFiles = 3;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    private async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        shouldRender = false;
        int maxFileSize = 1024 * 15;

```

```

var upload = false;

using var content = new MultipartFormDataContent();

foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
{
    if (uploadResults.SingleOrDefault(
        f => f.FileName == file.Name) is null)
    {
        try
        {
            files.Add(new() { Name = file.Name });

            var stream = new LazyBrowserFileStream(file,
maxFileSize);

            var fileContent = new StreamContent(stream);

            fileContent.Headers.ContentType =
                new MediaTypeHeaderValue(file.ContentType);

            content.Add(
                content: fileContent,
                name: "\"files\"",
                fileName: file.Name);

            upload = true;
        }
        catch (Exception ex)
        {
            Logger.LogInformation(
                "{FileName} not uploaded (Err: 6): {Message}",
                file.Name, ex.Message);

            uploadResults.Add(
                new()
                {
                    FileName = file.Name,
                    ErrorCode = 6,
                    Uploaded = false
                });
        }
    }
}

if (upload)
{
    var client = ClientFactory.CreateClient();

    var response =
        await client.PostAsync("https://localhost:5001/Filesave",
            content);

    if (response.IsSuccessStatusCode)
    {
        var options = new JsonSerializerOptions

```

```

        {
            PropertyNameCaseInsensitive = true,
        };

        using var responseStream =
            await response.Content.ReadAsStreamAsync();

        var newUploadResults = await JsonSerializer
            .DeserializeAsync<IList<UploadResult>>(responseStream,
options);

        if (newUploadResults is not null)
        {
            uploadResults =
uploadResults.Concat(newUploadResults).ToList();
        }
    }

    shouldRender = true;
}

private static bool FileUpload(IList<UploadResult> uploadResults,
    string? fileName, ILogger<FileUpload2> logger, out UploadResult
result)
{
    result = uploadResults.SingleOrDefault(f => f.FileName == fileName)
?? new();

    if (!result.Uploaded)
    {
        logger.LogInformation("{FileName} not uploaded (Err: 5)",
fileName);
        result.ErrorCode = 5;
    }

    return result.Uploaded;
}

private class File
{
    public string? Name { get; set; }
}
}

```

If the component limits file uploads to a single file at a time or if the component only adopts interactive client-side rendering (CSR, `InteractiveWebAssembly`), the component can avoid the use of the `LazyBrowserFileStream` and use a [Stream](#). The following demonstrates the changes for the `FileUpload2` component:

```
diff
```

```
- var stream = new LazyBrowserFileStream(file, maxFileSize);  
- var fileContent = new StreamContent(stream);  
+ var fileContent = new StreamContent(file.OpenReadStream(maxFileSize));
```

Remove the `LazyBrowserFileStream` class (`LazyBrowserFileStream.cs`), as it isn't used.

The following controller in the web API project saves uploaded files from the client.

Important

The controller in this section is intended for use in a separate web API project from the Blazor app. The web API should [mitigate Cross-Site Request Forgery \(XSRF/CSRF\) attacks](#) if file upload users are authenticated.

To use the following code, create a `Development/unsafe_uploads` folder at the root of the web API project for the app running in the `Development` environment.

Because the example uses the app's `environment` as part of the path where files are saved, additional folders are required if other environments are used in testing and production. For example, create a `Staging/unsafe_uploads` folder for the `Staging` environment. Create a `Production/unsafe_uploads` folder for the `Production` environment.

Warning

The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see [Upload files in ASP.NET Core](#).

`Controllers/FilesaveController.cs`:

C#

```
using System.Net;  
using Microsoft.AspNetCore.Mvc;  
  
[ApiController]  
[Route("[controller]")]  
public class FilesaveController(  
    IHostEnvironment env, ILogger<FilesaveController> logger)  
    : ControllerBase
```

```
[HttpPost]
public async Task<ActionResult<IList<UploadResult>>> PostFile(
    [FromForm] IEnumerable<IFormFile> files)
{
    var maxAllowedFiles = 3;
    long maxFileSize = 1024 * 15;
    var filesProcessed = 0;
    var resourcePath = new Uri($"{Request.Scheme}://{Request.Host}/");
    List<UploadResult> uploadResults = [];

    foreach (var file in files)
    {
        var uploadResult = new UploadResult();
        string trustedFileNameForFileStorage;
        var untrustedFileName = file.FileName;
        uploadResult.FileName = untrustedFileName;
        var trustedFileNameForDisplay =
            WebUtility.HtmlEncode(untrustedFileName);

        if (filesProcessed < maxAllowedFiles)
        {
            if (file.Length == 0)
            {
                logger.LogInformation("{FileName} length is 0 (Err: 1)",
                    trustedFileNameForDisplay);
                uploadResult.ErrorCode = 1;
            }
            else if (file.Length > maxFileSize)
            {
                logger.LogInformation("{FileName} of {Length} bytes is "
                    +
                    "larger than the limit of {Limit} bytes (Err: 2)",
                    trustedFileNameForDisplay, file.Length,
                    maxFileSize);
                uploadResult.ErrorCode = 2;
            }
            else
            {
                try
                {
                    trustedFileNameForFileStorage =
                        Path.GetRandomFileName();
                    var path = Path.Combine(env.ContentRootPath,
                        env.EnvironmentName, "unsafe_uploads",
                        trustedFileNameForFileStorage);

                    await using FileStream fs = new(path,
                        FileMode.Create);

                    await file.CopyToAsync(fs);

                    logger.LogInformation("{FileName} saved at {Path}",
                        trustedFileNameForDisplay, path);
                    uploadResult.Uploaded = true;
                    uploadResult.StoredFileName =
```

```

trustedFileNameForFileStorage;
    }
    catch (IOException ex)
    {
        logger.LogError("{FileName} error on upload (Err:
3): {Message}",
            trustedFileNameForDisplay, ex.Message);
        uploadResult.ErrorCode = 3;
    }

    filesProcessed++;
}
else
{
    logger.LogInformation("{FileName} not uploaded because the "
+
        "request exceeded the allowed {Count} of files (Err:
4)",
        trustedFileNameForDisplay, maxAllowedFiles);
    uploadResult.ErrorCode = 4;
}

uploadResults.Add(uploadResult);
}

return new CreatedResult(resourcePath, uploadResults);
}
}

```

In the preceding code, [GetRandomFileName](#) is called to generate a secure file name. Never trust the file name provided by the browser, as a cyberattacker may choose an existing file name that overwrites an existing file or send a path that attempts to write outside of the app.

The server app must register controller services and map controller endpoints. For more information, see [Routing to controller actions in ASP.NET Core](#).

Upload files to a server with client-side rendering (CSR)

This section applies to client-side rendered (CSR) components in Blazor Web Apps or Blazor WebAssembly apps.

The following example demonstrates uploading files to a backend web API controller in a separate app, possibly on a separate server, from a component in a Blazor Web App that adopts CSR or a component in a Blazor WebAssembly app.

The following `UploadResult` class maintains the result of an uploaded file. When a file fails to upload on the server, an error code is returned in `ErrorCode` for display to the user. A safe file name is generated on the server for each file and returned to the client in `StoredFileName` for display. Files are keyed between the client and server using the unsafe/untrusted file name in `FileName`.

`UploadResult.cs`:

C#

```
public class UploadResult
{
    public bool Uploaded { get; set; }
    public string? FileName { get; set; }
    public string? StoredFileName { get; set; }
    public int ErrorCode { get; set; }
}
```

ⓘ Note

The preceding `UploadResult` class can be shared between client- and server-based projects. When client and server projects share the class, add an import to each project's `_Imports.razor` file for the shared project. For example:

razor

```
@using BlazorSample.Shared
```

The following `FileUpload2` component:

- Permits users to upload files from the client.
- Displays the untrusted/unsafe file name provided by the client in the UI. The untrusted/unsafe file name is automatically HTML-encoded by Razor for safe display in the UI.

A security best practice for production apps is to avoid sending error messages to clients that might reveal sensitive information about an app, server, or network. Providing detailed error messages can aid a malicious user in devising attacks on an app, server, or network. The example code in this section only sends back an error code number (`int`) for display by the component client-side if a server-side error occurs. If a user requires assistance with a file upload, they provide the error code to support

personnel for support ticket resolution without ever knowing the exact cause of the error.

Warning

Don't trust file names supplied by clients for:

- Saving the file to a file system or service.
- Display in UIs that don't encode file names automatically or via developer code.

For more information on security considerations when uploading files to a server, see [Upload files in ASP.NET Core](#).

In the Blazor Web App main project, add [IHttpClientFactory](#) and related services in the project's `Program` file:

```
C#
```

```
builder.Services.AddHttpClient();
```

The `HttpClient` services must be added to the main project because the client-side component is prerendered on the server. If you [disable prerendering for the following component](#), you aren't required to provide the `HttpClient` services in the main app and don't need to add the preceding line to the main project.

For more information on adding `HttpClient` services to an ASP.NET Core app, see [Make HTTP requests using IHttpClientFactory in ASP.NET Core](#).

The client project (`.Client`) of a Blazor Web App must also register an `HttpClient` for HTTP POST requests to a backend web API controller. Confirm or add the following to the client project's `Program` file:

```
C#
```

```
builder.Services.AddScoped(sp =>  
    new HttpClient { BaseAddress = new  
    Uri(builder.HostEnvironment.BaseAddress) });
```

The preceding example sets the base address with

`builder.HostEnvironment.BaseAddress` ([IWebAssemblyHostEnvironment.BaseAddress](#)), which gets the base address for the app and is typically derived from the `<base>` tag's

`href` value in the host page. If you're calling an external web API, set the URI to the web API's base address.

Specify the Interactive WebAssembly render mode attribute at the top of the following component in a Blazor Web App:

razor

```
@rendermode InteractiveWebAssembly
```

FileUpload2.razor:

razor

```
@page "/file-upload-2"
@using System.Linq
@using System.Net.Http.Headers
@inject HttpClient Http
@inject ILogger<FileUpload2> Logger

<PageTitle>File Upload 2</PageTitle>

<h1>File Upload Example 2</h1>

<p>
    <label>
        Upload up to @maxAllowedFiles files:
        <InputFile OnChange="OnInputFileChange" multiple />
    </label>
</p>

@if (files.Count > 0)
{
    <div class="card">
        <div class="card-body">
            <ul>
                @foreach (var file in files)
                {
                    <li>
                        File: @file.Name
                        <br>
                        @if (FileUpload(uploadResults, file.Name, Logger,
                            out var result))
                        {
                            <span>
                                Stored File Name: @result.StoredFileName
                            </span>
                        }
                        else
                        {
                            <span>
```

```

        There was an error uploading the file
        (Error: @result.ErrorCode).
    </span>
    }
</li>
}
</ul>
</div>
</div>
}

@code {
    private List<File> files = new();
    private List<UploadResult> uploadResults = new();
    private int maxAllowedFiles = 3;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    private async Task OnInputChange(InputFileChangeEventArgs e)
    {
        shouldRender = false;
        long maxFileSize = 1024 * 15;
        var upload = false;

        using var content = new MultipartFormDataContent();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            if (uploadResults.SingleOrDefault(
                f => f.FileName == file.Name) is null)
            {
                try
                {
                    files.Add(new() { Name = file.Name });

                    var fileContent = new
StreamContent(file.OpenReadStream(maxFileSize));

                    fileContent.Headers.ContentType =
                        new MediaTypeHeaderValue(file.ContentType);

                    content.Add(
                        content: fileContent,
                        name: "\"files\"",
                        fileName: file.Name);

                    upload = true;
                }
                catch (Exception ex)
                {
                    Logger.LogInformation(
                        "{FileName} not uploaded (Err: 6): {Message}",
                        file.Name, ex.Message);
                }
            }
        }
    }
}

```

```

        uploadResults.Add(
            new()
            {
                FileName = file.Name,
                ErrorCode = 6,
                Uploaded = false
            });
    }
}

if (upload)
{
    var response = await Http.PostAsync("/Filesave", content);

    var newUploadResults = await response.Content
        .ReadFromJsonAsync<IList<UploadResult>>();

    if (newUploadResults is not null)
    {
        uploadResults =
uploadResults.Concat(newUploadResults).ToList();
    }

    shouldRender = true;
}

private static bool FileUpload(IList<UploadResult> uploadResults,
    string? fileName, ILogger<FileUpload2> logger, out UploadResult
result)
{
    result = uploadResults.SingleOrDefault(f => f.FileName == fileName)
?? new();

    if (!result.Uploaded)
    {
        logger.LogInformation("{FileName} not uploaded (Err: 5)",
fileName);
        result.ErrorCode = 5;
    }

    return result.Uploaded;
}

private class File
{
    public string? Name { get; set; }
}
}

```

The following controller in the server-side project saves uploaded files from the client.

To use the following code, create a `Development/unsafe_uploads` folder at the root of the server-side project for the app running in the `Development` environment.

Because the example uses the app's `environment` as part of the path where files are saved, additional folders are required if other environments are used in testing and production. For example, create a `Staging/unsafe_uploads` folder for the `Staging` environment. Create a `Production/unsafe_uploads` folder for the `Production` environment.

Warning

The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see [Upload files in ASP.NET Core](#).

In the following example, update the shared project's namespace to match the shared project if a shared project is supplying the `UploadResult` class.

`Controllers/FilesaveController.cs`:

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

[ApiController]
[Route("[controller]")]
public class FilesaveController(
    IHostEnvironment env, ILogger<FilesaveController> logger)
    : ControllerBase
{
    [HttpPost]
    public async Task<ActionResult<IList<UploadResult>>> PostFile(
        [FromForm] IEnumerable<IFormFile> files)
    {
        var maxAllowedFiles = 3;
        long maxFileSize = 1024 * 15;
        var filesProcessed = 0;
```

```

var resourcePath = new Uri($"{Request.Scheme}://{Request.Host}/");
List<UploadResult> uploadResults = [];

foreach (var file in files)
{
    var uploadResult = new UploadResult();
    string trustedFileNameForFileStorage;
    var untrustedFileName = file.FileName;
    uploadResult.FileName = untrustedFileName;
    var trustedFileNameForDisplay =
        WebUtility.HtmlEncode(untrustedFileName);

    if (filesProcessed < maxAllowedFiles)
    {
        if (file.Length == 0)
        {
            logger.LogInformation("{FileName} length is 0 (Err: 1)",
                trustedFileNameForDisplay);
            uploadResult.ErrorCode = 1;
        }
        else if (file.Length > maxFileSize)
        {
            logger.LogInformation("{FileName} of {Length} bytes is "
+
                "larger than the limit of {Limit} bytes (Err: 2)",
                trustedFileNameForDisplay, file.Length,
maxFileSize);
            uploadResult.ErrorCode = 2;
        }
        else
        {
            try
            {
                trustedFileNameForFileStorage =
Path.GetRandomFileName();
                var path = Path.Combine(env.ContentRootPath,
                    env.EnvironmentName, "unsafe_uploads",
                    trustedFileNameForFileStorage);

                await using FileStream fs = new(path,
FileMode.Create);
                await file.CopyToAsync(fs);

                logger.LogInformation("{FileName} saved at {Path}",
                    trustedFileNameForDisplay, path);
                uploadResult.Uploaded = true;
                uploadResult.StoredFileName =
trustedFileNameForFileStorage;
            }
            catch (IOException ex)
            {
                logger.LogError("{FileName} error on upload (Err:
3): {Message}",
                    trustedFileNameForDisplay, ex.Message);
                uploadResult.ErrorCode = 3;
            }
        }
    }
}

```

```

        }
    }

    filesProcessed++;
}
else
{
    logger.LogInformation("{FileName} not uploaded because the "
+
        "request exceeded the allowed {Count} of files (Err:
4)",
        trustedFileNameForDisplay, maxAllowedFiles);
    uploadResult.ErrorCode = 4;
}

uploadResults.Add(uploadResult);
}

return new CreatedResult(resourcePath, uploadResults);
}
}

```

In the preceding code, [GetRandomFileName](#) is called to generate a secure file name. Never trust the file name provided by the browser, as a cyberattacker may choose an existing file name that overwrites an existing file or send a path that attempts to write outside of the app.

The server app must register controller services and map controller endpoints. For more information, see [Routing to controller actions in ASP.NET Core](#).

Cancel a file upload

A file upload component can detect when a user has cancelled an upload by using a [CancellationToken](#) when calling into the [IBrowserFile.OpenReadStream](#) or [StreamReader.ReadAsync](#).

Create a [CancellationTokenSource](#) for the `InputFile` component. At the start of the `OnInputFileChange` method, check if a previous upload is in progress.

If a file upload is in progress:

- Call [Cancel](#) on the previous upload.
- Create a new [CancellationTokenSource](#) for the next upload and pass the [CancellationTokenSource.Token](#) to [OpenReadStream](#) or [ReadAsync](#).

Upload files server-side with progress

The following example demonstrates how to upload files in a server-side app with upload progress displayed to the user.

To use the following example in a test app:

- Create a folder to save uploaded files for the `Development` environment: `Development/unsafe_uploads`.
- Configure the maximum file size (`maxFileSize`, 15 KB in the following example) and maximum number of allowed files (`maxAllowedFiles`, 3 in the following example).
- Set the buffer to a different value (10 KB in the following example), if desired, for increased granularity in progress reporting. We don't recommended using a buffer larger than 30 KB due to performance and security concerns.

⚠ Warning

The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see [Upload files in ASP.NET Core](#).

`FileUpload3.razor`:

razor

```
@page "/file-upload-3"
@inject ILogger<FileUpload3> Logger
@inject IWebHostEnvironment Environment

<PageTitle>File Upload 3</PageTitle>

<h1>File Upload Example 3</h1>

<p>
    <label>
        Max file size:
        <input type="number" @bind="maxFileSize" />
    </label>
</p>

<p>
    <label>
        Max allowed files:
        <input type="number" @bind="maxAllowedFiles" />
    </label>
</p>
```

```

<p>
    <label>
        Upload up to @maxAllowedFiles of up to @maxFileSize bytes:
        <InputFile OnChange="LoadFiles" multiple />
    </label>
</p>

@if (isLoading)
{
    <p>Progress: @string.Format("{0:P0}", progressPercent)</p>
}
else
{
    <ul>
        @foreach (var file in loadedFiles)
        {
            <li>
                <ul>
                    <li>Name: @file.Name</li>
                    <li>Last modified: @file.LastModified.ToString()</li>
                    <li>Size (bytes): @file.Size</li>
                    <li>Content type: @file.ContentType</li>
                </ul>
            </li>
        }
    </ul>
}

@code {
    private List<IBrowserFile> loadedFiles = new();
    private long maxFileSize = 1024 * 15;
    private int maxAllowedFiles = 3;
    private bool isLoading;
    private decimal progressPercent;

    private async Task LoadFiles(InputFileChangeEventArgs e)
    {
        isLoading = true;
        loadedFiles.Clear();
        progressPercent = 0;

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            try
            {
                var trustedFileName = Path.GetRandomFileName();
                var path = Path.Combine(Environment.ContentRootPath,
                    Environment.EnvironmentName, "unsafe_uploads",
trustedFileName);

                await using FileStream writeStream = new(path,
 FileMode.Create);
                using var readStream = file.OpenReadStream(maxFileSize);
                var bytesRead = 0;
                var totalRead = 0;

```



```

        var buffer = new byte[1024 * 10];

        while ((bytesRead = await readStream.ReadAsync(buffer)) !=
0)
        {
            totalRead += bytesRead;
            await writeStream.WriteAsync(buffer, 0, bytesRead);
            progressPercent = Decimal.Divide(totalRead, file.Size);
            StateHasChanged();
        }

        loadedFiles.Add(file);

        Logger.LogInformation(
            "Unsafe Filename: {UnsafeFilename} File saved:
{Filename}",
            file.Name, trustedFileName);
    }
    catch (Exception ex)
    {
        Logger.LogError("File: {FileName} Error: {Error}",
            file.Name, ex.Message);
    }
}

    isLoading = false;
}
}

```

For more information, see the following API resources:

- [FileStream](#): Provides a [Stream](#) for a file, supporting both synchronous and asynchronous read and write operations.
- [FileStream.ReadAsync](#): The preceding `FileUpload3` component reads the stream asynchronously with [ReadAsync](#). Reading a stream synchronously with [Read](#) isn't supported in Razor components.

File streams

With server interactivity, file data is streamed over the SignalR connection into .NET code on the server as the file is read.

For a WebAssembly-rendered component, file data is streamed directly into the .NET code within the browser.

Upload image preview

For an image preview of uploading images, start by adding an `InputFile` component with a component reference and an `OnChange` handler:

razor

```
<InputFile @ref="inputFile" OnChange="ShowPreview" />
```

Add an image element with an [element reference](#), which serves as the placeholder for the image preview:

razor

```
<img @ref="previewImageElem" />
```

Add the associated references:

razor

```
@code {  
    private InputFile? inputFile;  
    private ElementReference previewImageElem;  
}
```

In JavaScript, add a function called with an HTML [input](#) and [img](#) element that performs the following:

- Extracts the selected file.
- Creates an object URL with [createUrl](#).
- Sets an event listener to revoke the object URL with [revokeObjectURL](#) after the image is loaded, so memory isn't leaked.
- Sets the `img` element's source to display the image.

JavaScript

```
window.previewImage = (inputElem, imgElem) => {  
    const url = URL.createObjectURL(inputElem.files[0]);  
    imgElem.addEventListener('load', () => URL.revokeObjectURL(url), { once:  
true });  
    imgElem.src = url;  
}
```

Finally, use an injected `IJSRuntime` to add the `OnChange` handler that calls the JavaScript function:

razor

```
@inject IJSRuntime JS

...

@code {
    ...

    private async Task ShowPreview() => await JS.InvokeVoidAsync(
        "previewImage", inputFile!.Element, previewImageElem);
}
```

The preceding example is for uploading a single image. The approach can be expanded to support `multiple` images.

The following `FileUpload4` component shows the complete example.

`FileUpload4.razor`:

razor

```
@page "/file-upload-4"
@inject IJSRuntime JS

<h1>File Upload Example</h1>

<InputFile @ref="inputFile" OnChange="ShowPreview" />

<img style="max-width:200px;max-height:200px" @ref="previewImageElem" />

@code {
    private InputFile? inputFile;
    private ElementReference previewImageElem;

    private async Task ShowPreview() => await JS.InvokeVoidAsync(
        "previewImage", inputFile!.Element, previewImageElem);
}
```

Upload files to an external service

Instead of an app handling file upload bytes and the app's server receiving uploaded files, clients can directly upload files to an external service. The app can safely process the files from the external service on demand. This approach hardens the app and its server against malicious attacks and potential performance problems.

Consider an approach that uses [Azure Files](#), [Azure Blob Storage](#), or a third-party service with the following potential benefits:

- Upload files from the client directly to an external service with a JavaScript client library or REST API. For example, Azure offers the following client libraries and APIs:
 - [Azure Storage File Share client library](#)
 - [Azure Files REST API](#)
 - [Azure Storage Blob client library for JavaScript](#)
 - [Blob service REST API](#)
- Authorize user uploads with a user-delegated shared-access signature (SAS) token generated by the app (server-side) for each client file upload. For example, Azure offers the following SAS features:
 - [Azure Storage File Share client library for JavaScript: with SAS Token](#)
 - [Azure Storage Blob client library for JavaScript: with SAS Token](#)
- Provide automatic redundancy and file share backup.
- Limit uploads with quotas. Note that Azure Blob Storage's quotas are set at the account level, not the container level. However, Azure Files quotas are at the file share level and might provide better control over upload limits. For more information, see the Azure documents linked earlier in this list.
- Secure files with server-side encryption (SSE).

For more information on Azure Blob Storage and Azure Files, see the [Azure Storage documentation](#).

Server-side SignalR message size limit

File uploads may fail even before they start, when Blazor retrieves data about the files that exceeds the maximum SignalR message size.

SignalR defines a message size limit that applies to every message Blazor receives, and the [InputFile](#) component streams files to the server in messages that respect the configured limit. However, the first message, which indicates the set of files to upload, is sent as a unique single message. The size of the first message may exceed the SignalR message size limit. The issue isn't related to the size of the files, it's related to the number of files.

The logged error is similar to the following:

```
Error: Connection disconnected with error 'Error: Server returned an error on close:
Connection closed with an error.'. e.log @ blazor.server.js:1
```

When uploading files, reaching the message size limit on the first message is rare. If the limit is reached, the app can configure `HubOptions.MaximumReceiveMessageSize` with a larger value.

For more information on SignalR configuration and how to set `MaximumReceiveMessageSize`, see [ASP.NET Core Blazor SignalR guidance](#).

Maximum parallel invocations per client hub setting

Blazor relies on `MaximumParallelInvocationsPerClient` set to 1, which is the default value.

Increasing the value leads to a high probability that `CopyTo` operations throw `System.InvalidOperationException: 'Reading is not allowed after reader was completed.'`. For more information, see [MaximumParallelInvocationsPerClient > 1 breaks file upload in Blazor Server mode \(dotnet/aspnetcore #53951\)](#).

Troubleshoot

The line that calls `IBrowserFile.OpenReadStream` throws a `System.TimeoutException`:

```
System.TimeoutException: Did not receive any data in the allotted time.
```

Possible causes:

- Using the [Autofac Inversion of Control \(IoC\) container](#) instead of the built-in ASP.NET Core dependency injection container. To resolve the issue, set `DisableImplicitFromServicesParameters` to `true` in the [server-side circuit handler hub options](#). For more information, see [FileUpload: Did not receive any data in the allotted time \(dotnet/aspnetcore #38842\)](#).
- Not reading the stream to completion. This isn't a framework issue. Trap the exception and investigate it further in your local environment/network.
- Using server-side rendering and calling `OpenReadStream` on multiple files before reading them to completion. To resolve the issue, use the `LazyBrowserFileStream` class and approach described in the [Upload files to a server with server-side rendering](#) section of this article.

Additional resources

- [ASP.NET Core Blazor file downloads](#)
- [Upload files in ASP.NET Core](#)
- [ASP.NET Core Blazor forms overview](#)
- [Blazor samples GitHub repository \(dotnet/blazor-samples\) ↗](#) (how to download)

ASP.NET Core Blazor file downloads

Article • 11/06/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to download files in Blazor apps.

File downloads

This article covers approaches for the following scenarios, where a file shouldn't be opened by a browser but downloaded and saved on the client:

- [Stream file content to a raw binary data buffer on the client](#): Typically, this approach is used for relatively small files (< 250 MB).
- [Download a file via a URL without streaming](#): Usually, this approach is used for relatively large files (> 250 MB).

When downloading files from a different origin than the app, Cross-Origin Resource Sharing (CORS) considerations apply. For more information, see the [Cross-Origin Resource Sharing \(CORS\)](#) section.

Security considerations

Use caution when providing users with the ability to download files from a server. Cyberattackers may execute [Denial of Service \(DoS\)](#) attacks, [API exploitation attacks](#)[↗], or attempt to compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Download files from a dedicated file download area on the server, preferably from a non-system drive. Using a dedicated location makes it easier to impose security restrictions on downloadable files. Disable execute permissions on the file download area.

- Client-side security checks are easy to circumvent by malicious users. Always perform client-side security checks on the server, too.
- Don't receive files from users or other untrusted sources and then make the files available for immediate download without performing security checks on the files. For more information, see [Upload files in ASP.NET Core](#).

Download from a stream

This section applies to files that are typically up to 250 MB in size.

The recommended approach for downloading relatively small files (< 250 MB) is to stream file content to a raw binary data buffer on the client with [JavaScript \(JS\) interop](#). This approach is effective for components that adopt an interactive render mode but not components that adopt static server-side rendering (static SSR).

Warning

The approach in this section reads the file's content into a [JS ArrayBuffer](#). This approach loads the entire file into the client's memory, which can impair performance. To download relatively large files (>= 250 MB), we recommend following the guidance in the [Download from a URL](#) section.

The following `downloadFileFromStream` JS function:

- Reads the provided stream into an [ArrayBuffer](#).
- Creates a [Blob](#) to wrap the `ArrayBuffer`.
- Creates an object URL to serve as the file's download address.
- Creates an [HTMLAnchorElement](#) (`<a>` element).
- Assigns the file's name (`fileName`) and URL (`url`) for the download.
- Triggers the download by firing a [click event](#) on the anchor element.
- Removes the anchor element.
- Revokes the object URL (`url`) by calling [URL.revokeObjectURL](#). **This is an important step to ensure memory isn't leaked on the client.**

HTML

```
<script>
window.downloadFileFromStream = async (fileName, contentStreamReference)
=> {
  const arrayBuffer = await contentStreamReference.arrayBuffer();
  const blob = new Blob([arrayBuffer]);
  const url = URL.createObjectURL(blob);
  const anchorElement = document.createElement('a');
```



```

        anchorElement.href = url;
        anchorElement.download = fileName ?? '';
        anchorElement.click();
        anchorElement.remove();
        URL.revokeObjectURL(url);
    }
</script>

```

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

The following component:

- Uses native byte-streaming interop to ensure efficient transfer of the file to the client.
- Has a method named `GetFileStream` to retrieve a [Stream](#) for the file that's downloaded to clients. Alternative approaches include retrieving a file from storage or generating a file dynamically in C# code. For this demonstration, the app creates a 50 KB file of random data from a new byte array (`new byte[]`). The bytes are wrapped with a [MemoryStream](#) to serve as the example's dynamically-generated binary file.
- The `DownloadFileFromStream` method:
 - Retrieves the [Stream](#) from `GetFileStream`.
 - Specifies a file name when file is saved on the user's machine. The following example names the file `quote.txt`.
 - Wraps the [Stream](#) in a [DotNetStreamReference](#), which allows streaming the file data to the client.
 - Invokes the `downloadFileFromStream` JS function to accept the data on the client.

`FileDownload1.razor`:

razor

```

@page "/file-download-1"
@using System.IO
@inject IJSRuntime JS

<PageTitle>File Download 1</PageTitle>

<h1>File Download Example 1</h1>

<button @onclick="DownloadFileFromStream">
    Download File From Stream

```

```

</button>

@code {
    private Stream GetFileStream()
    {
        var randomBinaryData = new byte[50 * 1024];
        var fileStream = new MemoryStream(randomBinaryData);

        return fileStream;
    }

    private async Task DownloadFileFromStream()
    {
        var fileStream = GetFileStream();
        var fileName = "log.bin";

        using var streamRef = new DotNetStreamReference(stream: fileStream);

        await JS.InvokeVoidAsync("downloadFileFromStream", fileName,
streamRef);
    }
}

```

For a component in a server-side app that must return a [Stream](#) for a physical file, the component can call [File.OpenRead](#), as the following example demonstrates:

C#

```
private Stream GetFileStream() => File.OpenRead(@"{PATH}");
```

In the preceding example, the `{PATH}` placeholder is the path to the file. The `@` prefix indicates that the string is a *verbatim string literal*, which permits the use of backslashes (`\`) in a Windows OS path and embedded double-quotes (`"`) for a single quote in the path. Alternatively, avoid the string literal (`@`) and use either of the following approaches:

- Use escaped backslashes (`\\`) and quotes (`"`).
- Use forward slashes (`/`) in the path, which are supported across platforms in ASP.NET Core apps, and escaped quotes (`"`).

Download from a URL

This section applies to files that are relatively large, typically 250 MB or larger.

The recommended approach for downloading relatively large files (≥ 250 MB) with interactively-rendered components or files of any size for statically-rendered

components is to use JS to trigger an anchor element with the file's name and URL.

The example in this section uses a download file named `quote.txt`, which is placed in a folder named `files` in the app's web root (`wwwroot` folder). The use of the `files` folder is only for demonstration purposes. You can organize downloadable files in any folder layout within the web root (`wwwroot` folder) that you prefer, including serving the files directly from the `wwwroot` folder.

`wwwroot/files/quote.txt`:

text

When victory is ours, we'll wipe every trace of the Thals and their city from the face of this land. We will avenge the deaths of all Kaleds who've fallen in the cause of right and justice and build a peace which will be a monument to their sacrifice. Our battle cry will be "Total extermination of the Thals!"

- General Ravon (Guy Siner, <http://guysiner.com/>)
Dr. Who: Genesis of the Daleks (<https://www.bbc.co.uk/programmes/p00vd5g2>)
Copyright 1975 BBC (<https://www.bbc.co.uk/>)

The following `triggerFileDownload` JS function:

- Creates an [HTMLAnchorElement](#) (`<a>` element).
- Assigns the file's name (`fileName`) and URL (`url`) for the download.
- Triggers the download by firing a [click event](#) on the anchor element.
- Removes the anchor element.

HTML

```
<script>
  window.triggerFileDownload = (fileName, url) => {
    const anchorElement = document.createElement('a');
    anchorElement.href = url;
    anchorElement.download = fileName ?? '';
    anchorElement.click();
    anchorElement.remove();
  }
</script>
```

⚠ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

The following example component downloads the file from the same origin that the app uses. If the file download is attempted from a different origin, configure Cross-Origin Resource Sharing (CORS). For more information, see the [Cross-Origin Resource Sharing \(CORS\)](#) section.

FileDownload2.razor:

razor

```
@page "/file-download-2"
@inject IJSRuntime JS

<PageTitle>File Download 2</PageTitle>

<h1>File Download Example 2</h1>

<button @onclick="DownloadFileFromURL">
    Download File From URL
</button>

@code {
    private async Task DownloadFileFromURL()
    {
        var fileName = "quote.txt";
        var fileURL = "/files/quote.txt";
        await JS.InvokeVoidAsync("triggerFileDownload", fileName, fileURL);
    }
}
```

For interactive components, the button in the preceding example calls the `DownloadFileFromURL` handler to invoke the JavaScript (JS) function `triggerFileDownload`.

If the component adopts static server-side rendering (static SSR), add an event handler for the button ([addEventListener \(MDN documentation\)](#) [↗]) to call `triggerFileDownload` following the guidance in [ASP.NET Core Blazor JavaScript with static server-side rendering \(static SSR\)](#).

Cross-Origin Resource Sharing (CORS)

Without taking further steps to enable [Cross-Origin Resource Sharing \(CORS\)](#) [↗] for files that don't have the same origin as the app, downloading files won't pass CORS checks made by the browser.

For more information on CORS with ASP.NET Core apps and other Microsoft products and services that host files for download, see the following resources:

- [Enable Cross-Origin Requests \(CORS\) in ASP.NET Core](#)
- [Using Azure CDN with CORS \(Azure documentation\)](#)
- [Cross-Origin Resource Sharing \(CORS\) support for Azure Storage \(REST documentation\)](#)
- [Core Cloud Services - Set up CORS for your website and storage assets \(Learn module\)](#)
- [IIS CORS module Configuration Reference \(IIS documentation\)](#)

Additional resources

- [ASP.NET Core Blazor static files](#)
- [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#)
- [JavaScript location in ASP.NET Core Blazor apps](#)
- [ASP.NET Core Blazor JavaScript with static server-side rendering \(static SSR\)](#)
- [<a>: The Anchor element: Security and privacy \(MDN documentation\)](#) [↗](#)
- [ASP.NET Core Blazor file uploads](#)
- [Blazor samples GitHub repository \(dotnet/blazor-samples\)](#) [↗](#) (how to download)

ASP.NET Core Blazor JavaScript interoperability (JS interop)

Article • 11/19/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

A Blazor app can invoke JavaScript (JS) functions from .NET methods and .NET methods from JS functions. These scenarios are called *JavaScript interoperability (JS interop)*.

Further JS interop guidance is provided in the following articles:

- [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#)
- [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#)

Note

JavaScript `[JSImport]` / `[JSEExport]` interop API is available for client-side components in ASP.NET Core in .NET 7 or later.

For more information, see [JavaScript JSImport/JSEExport interop with ASP.NET Core Blazor](#).

Compression for interactive server components with untrusted data

With compression, which is enabled by default, avoid creating secure (authenticated/authorized) interactive server-side components that render data from untrusted sources. Untrusted sources include route parameters, query strings, data from JS interop, and any other source of data that a third-party user can control (databases, external services). For more information, see [ASP.NET Core Blazor SignalR guidance](#) and [Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering](#).

JavaScript interop abstractions and features package

The [@microsoft/dotnet-js-interop package \(npmjs.com\)](https://www.npmjs.com/package/@microsoft/dotnet-js-interop) [↗](#) ([Microsoft.JSInterop NuGet package](#) [↗](#)) provides abstractions and features for interop between .NET and JavaScript (JS) code. Reference source is available in the [dotnet/aspnetcore GitHub repository \(/src/JSInterop folder\)](#) [↗](#). For more information, see the GitHub repository's `README.md` file.

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#) [↗](#).

Additional resources for writing JS interop scripts in TypeScript:

- [TypeScript](#) [↗](#)
- [Tutorial: Create an ASP.NET Core app with TypeScript in Visual Studio](#)
- [Manage npm packages in Visual Studio](#)

Interaction with the DOM

Only mutate the DOM with JavaScript (JS) when the object doesn't interact with Blazor. Blazor maintains representations of the DOM and interacts directly with DOM objects. If an element rendered by Blazor is modified externally using JS directly or via JS Interop, the DOM may no longer match Blazor's internal representation, which can result in undefined behavior. Undefined behavior may merely interfere with the presentation of elements or their functions but may also introduce security risks to the app or server.

This guidance not only applies to your own JS interop code but also to any JS libraries that the app uses, including anything provided by a third-party framework, such as [Bootstrap JS](#) [↗](#) and [jQuery](#) [↗](#).

In a few documentation examples, JS interop is used to mutate an element *purely for demonstration purposes* as part of an example. In those cases, a warning appears in the text.

For more information, see [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#).

JavaScript class with a field of type function

A JavaScript class with a field of type function isn't supported by Blazor JS interop. Use Javascript functions in classes.

✗ `GreetingHelpers.sayHello` in the following class as a field of type function isn't discovered by Blazor's JS interop and can't be executed from C# code:

JavaScript

```
export class GreetingHelpers {  
    sayHello = function() {  
        ...  
    }  
}
```

✓ `GreetingHelpers.sayHello` in the following class as a function is supported:

JavaScript

```
export class GreetingHelpers {  
    sayHello() {  
        ...  
    }  
}
```

Arrow functions are also supported:

JavaScript

```
export class GreetingHelpers {  
    sayHello = () => {  
        ...  
    }  
}
```

Avoid inline event handlers

A JavaScript function can be invoked directly from an inline event handler. In the following example, `alertUser` is a JavaScript function called when the button is selected by the user:

HTML

```
<button onclick="alertUser">Click Me!</button>
```

However, the use of inline event handlers is a *poor design choice* for calling JavaScript functions:

- Mixing HTML markup and JavaScript code often leads to unmaintainable code.
- Inline event handler execution may be blocked by a [Content Security Policy \(CSP\)](#) (MDN documentation) [↗](#).

We recommend avoiding inline event handlers in favor of approaches that assign handlers in JavaScript with [addEventListener](#) [↗](#), as the following example demonstrates:

AlertUser.razor.js:

JavaScript

```
export function alertUser() {
    alert('The button was selected!');
}

export function addHandlers() {
    const btn = document.getElementById("btn");
    btn.addEventListener("click", alertUser);
}
```

AlertUser.razor:

razor

```
@page "/alert-user"
@implements IAsyncDisposable
@inject IJSRuntime JS

<h1>Alert User</h1>

<p>
    <button id="btn">Click Me!</button>
</p>

@code {
    private IJSObjectReference? module;

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
```

```

        "./Components/Pages/AlertUser.razor.js");

        await module.InvokeVoidAsync("addHandlers");
    }
}

async ValueTask IAsyncDisposable.DisposeAsync()
{
    if (module is not null)
    {
        try
        {
            await module.DisposeAsync();
        }
        catch (JSDisconnectedException)
        {
        }
    }
}
}
}

```

In the preceding example, `JSDisconnectedException` is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try - catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

For more information, see the following resources:

- [JavaScript location in ASP.NET Core Blazor apps](#)
- [Introduction to events \(MDN documentation\)](#) 

Asynchronous JavaScript calls

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side rendering models. When adopting server-side rendering, JS interop calls must be asynchronous because they're sent over a network connection. For apps that exclusively adopt client-side rendering, synchronous JS interop calls are supported.

For more information, see the following articles:

- [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#)
- [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#)

Object serialization

Blazor uses [System.Text.Json](#) for serialization with the following requirements and default behaviors:

- Types must have a default constructor, [get/set accessors](#) must be public, and fields are never serialized.
- Global default serialization isn't customizable to avoid breaking existing component libraries, impacts on performance and security, and reductions in reliability.
- Serializing .NET member names results in lowercase JSON key names.
- JSON is deserialized as [JsonElement](#) C# instances, which permit mixed casing. Internal casting for assignment to C# model properties works as expected in spite of any case differences between JSON key names and C# property names.
- Complex framework types, such as [KeyValuePair](#), might be [trimmed away by the IL Trimmer on publish](#) and not present for JS interop. We recommend creating custom types for types that the IL Trimmer trims away.
- Blazor always relies on [reflection for JSON serialization](#), including when using C# [source generation](#). Setting `JsonSerializerIsReflectionEnabledByDefault` to `false` in the app's project file results in an error when serialization is attempted.


[JsonConverter](#) API is available for custom serialization. Properties can be annotated with a [\[JsonConverter\] attribute](#) to override default serialization for an existing data type.

For more information, see the following resources in the .NET documentation:

- [JSON serialization and deserialization \(marshalling and unmarshalling\) in .NET](#)
- [How to customize property names and values with System.Text.Json](#)
- [How to write custom converters for JSON serialization \(marshalling\) in .NET](#)

Blazor supports optimized byte array JS interop that avoids encoding/decoding byte arrays into Base64. The app can apply custom serialization and pass the resulting bytes. For more information, see [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#).

DOM cleanup tasks during component disposal

Don't execute JS interop code for DOM cleanup tasks during component disposal. Instead, use the [MutationObserver](#)  pattern in JavaScript (JS) on the client for the following reasons:

- The component may have been removed from the DOM by the time your cleanup code executes in `Dispose{Async}`.
- During server-side rendering, the Blazor renderer may have been disposed by the framework by the time your cleanup code executes in `Dispose{Async}`.

The [MutationObserver](#) pattern allows you to run a function when an element is removed from the DOM.

In the following example, the `DOMCleanup` component:

- Contains a `<div>` with an `id` of `cleanupDiv`. The `<div>` element is removed from the DOM along with the rest of the component's DOM markup when the component is removed from the DOM.
- Loads the `DOMCleanup` JS class from the `DOMCleanup.razor.js` file and calls its `createObserver` function to set up the `MutationObserver` callback. These tasks are accomplished in the [OnAfterRenderAsync](#) lifecycle method.

`DOMCleanup.razor`:

razor

```
@page "/dom-cleanup"
@implements IAsyncDisposable
@inject IJSRuntime JS

<h1>DOM Cleanup Example</h1>

<div id="cleanupDiv"></div>

@code {
    private IJSObjectReference? module;

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

            await module.InvokeVoidAsync("DOMCleanup.createObserver");
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {

```

```

        await module.DisposeAsync();
    }
    catch (JSDisconnectedException)
    {
    }
}
}
}

```

In the preceding example, `JSDisconnectedException` is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try-catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

In the following example, the `MutationObserver` callback is executed each time a DOM change occurs. Execute your cleanup code when the `if` statement confirms that the target element (`cleanupDiv`) was removed (`if (targetRemoved) { ... }`). It's important to disconnect and delete the `MutationObserver` to avoid a memory leak after your cleanup code executes.

`DOMCleanup.razor.js` placed side-by-side with the preceding `DOMCleanup` component:

JavaScript

```

export class DOMCleanup {
    static observer;

    static createObserver() {
        const target = document.querySelector('#cleanupDiv');

        this.observer = new MutationObserver(function (mutations) {
            const targetRemoved = mutations.some(function (mutation) {
                const nodes = Array.from(mutation.removedNodes);
                return nodes.indexOf(target) !== -1;
            });

            if (targetRemoved) {
                // Cleanup resources here
                // ...

                // Disconnect and delete MutationObserver
                this.observer && this.observer.disconnect();
                delete this.observer;
            }
        });

        this.observer.observe(target.parentNode, { childList: true });
    }
}

```

```
}  
}  
  
window.DOMCleanup = DOMCleanup;
```

JavaScript interop calls without a circuit

This section only applies to server-side apps.

JavaScript (JS) interop calls can't be issued after Blazor's SignalR circuit is disconnected. Without a circuit during component disposal or at any other time that a circuit doesn't exist, the following method calls fail and log a message that the circuit is disconnected as a [JSDisconnectedException](#):

- JS interop method calls
 - [IJSRuntime.InvokeAsync](#)
 - [JSRuntimeExtensions.InvokeAsync](#)
 - [JSRuntimeExtensions.InvokeVoidAsync](#)
- `Dispose`/`DisposeAsync` calls on any [IJSObjectReference](#).

In order to avoid logging [JSDisconnectedException](#) or to log custom information, catch the exception in a [try-catch](#) statement.

For the following component disposal example:

- The server-side component implements [IAsyncDisposable](#).
- `module` is an [IJSObjectReference](#) for a JS module.
- [JSDisconnectedException](#) is caught and not logged.
- Optionally, you can log custom information in the `catch` statement at whatever log level you prefer. The following example doesn't log custom information because it assumes the developer doesn't care about when or where circuits are disconnected during component disposal.

```
C#  
  
async ValueTask IAsyncDisposable.DisposeAsync()  
{  
    try  
    {  
        if (module is not null)  
        {  
            await module.DisposeAsync();  
        }  
    }  
    catch (JSDisconnectedException)
```

```
{  
  }  
}
```

If you must clean up your own JS objects or execute other JS code on the client after a circuit is lost in a server-side Blazor app, use the [MutationObserver](#) pattern in JS on the client. The [MutationObserver](#) pattern allows you to run a function when an element is removed from the DOM.

For more information, see the following articles:

- [Handle errors in ASP.NET Core Blazor apps](#): The *JavaScript interop* section discusses error handling in JS interop scenarios.
- [ASP.NET Core Razor component lifecycle](#): The *Component disposal with `IDisposable` and `IAsyncDisposable`* section describes how to implement disposal patterns in Razor components.

Cached JavaScript files

JavaScript (JS) files and other static assets aren't generally cached on clients during development in the [Development environment](#). During development, static asset requests include the [Cache-Control header](#) with a value of [no-cache](#) or [max-age](#) with a value of zero (0).

During production in the [Production environment](#), JS files are usually cached by clients.

To disable client-side caching in browsers, developers usually adopt one of the following approaches:

- Disable caching when the browser's developer tools console is open. Guidance can be found in the developer tools documentation of each browser maintainer:
 - [Chrome DevTools](#)
 - [Microsoft Edge Developer Tools overview](#)
- Perform a manual browser refresh of any webpage of the Blazor app to reload JS files from the server. ASP.NET Core's HTTP Caching Middleware always honors a valid no-cache [Cache-Control header](#) sent by a client.

For more information, see:

- [ASP.NET Core Blazor environments](#)
- [Response caching in ASP.NET Core](#)

Size limits on JavaScript interop calls

This section only applies to interactive components in server-side apps. For client-side components, the framework doesn't impose a limit on the size of JavaScript (JS) interop inputs and outputs.

For interactive components in server-side apps, JS interop calls passing data from the client to the server are limited in size by the maximum incoming SignalR message size permitted for hub methods, which is enforced by

[HubOptions.MaximumReceiveMessageSize](#) (default: 32 KB). JS to .NET SignalR messages larger than [MaximumReceiveMessageSize](#) throw an error. The framework doesn't impose a limit on the size of a SignalR message from the hub to a client. For more information on the size limit, error messages, and guidance on dealing with message size limits, see [ASP.NET Core Blazor SignalR guidance](#).

Determine where the app is running

If it's relevant for the app to know where code is running for JS interop calls, use [OperatingSystem.IsBrowser](#) to determine if the component is executing in the context of browser on WebAssembly.

JavaScript location in ASP.NET Core Blazor apps

Article • 11/19/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

Load JavaScript (JS) code using any of the following approaches:

- [Load a script in <head> markup](#) (*Not generally recommended*)
- [Load a script in <body> markup](#)
- [Load a script from an external JavaScript file \(.js\) collocated with a component](#)
- [Load a script from an external JavaScript file \(.js\)](#)
- [Inject a script before or after Blazor starts](#)

Warning

Only place a `<script>` tag in a component file (`.razor`) if the component is guaranteed to adopt **static server-side rendering (static SSR)**, because the `<script>` tag can't be updated dynamically.

Note

Documentation examples usually place scripts in a `<script>` tag or load global scripts from external files. These approaches pollute the client with global functions. For production apps, we recommend placing JS into separate **JS modules** [↗](#) that can be imported when needed. For more information, see the **JavaScript isolation in JavaScript modules** section.

Load a script in `<head>` markup

The approach in this section isn't generally recommended.

Place the JavaScript (JS) tags (`<script>...</script>`) in the `<head>` element markup:

HTML

```
<head>
  ...

  <script>
    window.jsMethod = (methodParameter) => {
      ...
    };
  </script>
</head>
```

Loading JS from the `<head>` isn't the best approach for the following reasons:

- JS interop may fail if the script depends on Blazor. We recommend loading scripts using one of the other approaches, not via the `<head>` markup.
- The page may become interactive slower due to the time it takes to parse the JS in the script.

Load a script in `<body>` markup

Place the JavaScript tags (`<script>...</script>`) inside the closing `</body>` element after the Blazor script reference:

HTML

```
<body>
  ...

  <script src="{BLAZOR SCRIPT}"></script>
  <script>
    window.jsMethod = (methodParameter) => {
      ...
    };
  </script>
</body>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see [ASP.NET Core Blazor project structure](#).

Load a script from an external JavaScript file (`.js`) collocated with a component

Collocation of JavaScript (JS) files for Razor components is a convenient way to organize scripts in an app.

Razor components of Blazor apps collocate JS files using the `.razor.js` extension and are publicly addressable using the path to the file in the project:

```
{PATH}/{COMPONENT}.razor.js
```

- The `{PATH}` placeholder is the path to the component.
- The `{COMPONENT}` placeholder is the component.

When the app is published, the framework automatically moves the script to the web root. Scripts are moved to `bin/Release/{TARGET FRAMEWORK MONIKER}/publish/wwwroot/{PATH}/{COMPONENT}.razor.js`, where the placeholders are:

- `{TARGET FRAMEWORK MONIKER}` is the [Target Framework Moniker \(TFM\)](#).
- `{PATH}` is the path to the component.
- `{COMPONENT}` is the component name.

No change is required to the script's relative URL, as Blazor takes care of placing the JS file in published static assets for you.

This section and the following examples are primarily focused on explaining JS file collocation. The first example demonstrates a collocated JS file with an ordinary JS function. The second example demonstrates the use of a module to load a function, which is the recommended approach for most production apps. Calling JS from .NET is fully covered in [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#), where there are further explanations of the Blazor JS API with additional examples. Component disposal, which is present in the second example, is covered in [ASP.NET Core Razor component lifecycle](#).

The following `JsCollocation1` component loads a script via a [HeadContent component](#) and calls a JS function with `IJSRuntime.InvokeAsync`. The `{PATH}` placeholder is the path to the component.

Important

If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component (example: `Components/Pages` in .NET 8 or later or `Pages` in .NET 7 or earlier). In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

Add the following script after the Blazor script ([location of the Blazor start script](#)):

HTML

```
<script src="{PATH}/JsCollocation1.razor.js"></script>
```

JsCollocation1 component ({PATH}/JsCollocation1.razor):

razor

```
@page "/js-collocation-1"
@inject IJSRuntime JS

<PageTitle>JS Collocation 1</PageTitle>

<h1>JS Collocation Example 1</h1>

<button @onclick="ShowPrompt">Call showPrompt1</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}

@code {
    private string? result;

    public async void ShowPrompt()
    {
        result = await JS.InvokeAsync<string>(
            "showPrompt1", "What's your name?");
        StateHasChanged();
    }
}
```

The collocated JS file is placed next to the JsCollocation1 component file with the file name JsCollocation1.razor.js. In the JsCollocation1 component, the script is referenced at the path of the collocated file. In the following example, the showPrompt1 function accepts the user's name from a [Window prompt\(\)](#) and returns it to the JsCollocation1 component for display.

{PATH}/JsCollocation1.razor.js:

JavaScript

```
function showPrompt1(message) {
    return prompt(message, 'Type your name here');
```

```
}
```

The preceding approach isn't recommended for general use in production apps because the approach pollutes the client with global functions. A better approach for production apps is to use JS modules. The same general principles apply to loading a JS module from a collocated JS file, as the next example demonstrates.

The following `JsCollocation2` component's `OnAfterRenderAsync` method loads a JS module into `module`, which is an `IJSObjectReference` of the component class. `module` is used to call the `showPrompt2` function. The `{PATH}` placeholder is the path to the component.

❗ Important

If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component. In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

`JsCollocation2` component (`{PATH}/JsCollocation2.razor`):

razor

```
@page "/js-collocation-2"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>JS Collocation 2</PageTitle>

<h1>JS Collocation Example 2</h1>

<button @onclick="ShowPrompt">Call showPrompt2</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}

@code {
    private IJSObjectReference? module;
    private string? result;

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
```

```

    {
        /*
of          Change the {PATH} placeholder in the next line to the path
            the collocated JS file in the app. Examples:

            ./Components/Pages/JsCollocation2.razor.js (.NET 8 or later)
            ./Pages/JsCollocation2.razor.js (.NET 7 or earlier)
        */
        module = await JS.InvokeAsync<IJSObjectReference>("import",
            "{PATH}/JsCollocation2.razor.js");
    }
}

public async void ShowPrompt()
{
    if (module is not null)
    {
        result = await module.InvokeAsync<string>(
            "showPrompt2", "What's your name?");
        StateHasChanged();
    }
}

async ValueTask IAsyncDisposable.DisposeAsync()
{
    if (module is not null)
    {
        try
        {
            await module.DisposeAsync();
        }
        catch (JSDisconnectedException)
        {
        }
    }
}
}
}

```

In the preceding example, `JSDisconnectedException` is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try - catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

`{PATH}/JsCollocation2.razor.js`:

JavaScript

```
export function showPrompt2(message) {  
    return prompt(message, 'Type your name here');  
}
```

Use of scripts and modules for collocated JS in a Razor class library (RCL) is only supported for Blazor's JS interop mechanism based on the [IJSRuntime](#) interface. If you're implementing [JavaScript \[JSImport\]/\[JSExport\] interop](#), see [JavaScript JSImport/JSExport interop with ASP.NET Core Blazor](#).

For scripts or modules provided by a Razor class library (RCL) using [IJSRuntime](#)-based JS interop, the following path is used:

```
./_content/{PACKAGE ID}/{PATH}/{COMPONENT}.{EXTENSION}.js
```

- The path segment for the current directory (./) is required in order to create the correct static asset path to the JS file.
- The {PACKAGE ID} placeholder is the RCL's package identifier (or library name for a class library referenced by the app).
- The {PATH} placeholder is the path to the component. If a Razor component is located at the root of the RCL, the path segment isn't included.
- The {COMPONENT} placeholder is the component name.
- The {EXTENSION} placeholder matches the extension of component, either razor or cshtml.

In the following Blazor app example:

- The RCL's package identifier is AppJS.
- A module's scripts are loaded for the JsCollocation3 component (JsCollocation3.razor).
- The JsCollocation3 component is in the Components/Pages folder of the RCL.

C#

```
module = await JS.InvokeAsync<IJSObjectReference>("import",  
    "./_content/AppJS/Components/Pages/JsCollocation3.razor.js");
```

For more information on RCLs, see [Consume ASP.NET Core Razor components from a Razor class library \(RCL\)](#).

Load a script from an external JavaScript file (.js)

Place the JavaScript (JS) tags (`<script>...</script>`) with a script source (`src`) path inside the [closing `</body>` element](#) after the Blazor script reference:

HTML

```
<body>
    ...

    <script src="{BLAZOR SCRIPT}"></script>
    <script src="{SCRIPT PATH AND FILE NAME (.js)}"></script>
</body>
```

For the placeholders in the preceding example:

- The `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see [ASP.NET Core Blazor project structure](#).
- The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and script file name under `wwwroot`.

In the following example of the preceding `<script>` tag, the `scripts.js` file is in the `wwwroot/js` folder of the app:

HTML

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

You can also serve scripts directly from the `wwwroot` folder if you prefer not to keep all of your scripts in a separate folder under `wwwroot`:

HTML

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

When the external JS file is supplied by a [Razor class library](#), specify the JS file using its stable static web asset path: `_content/{PACKAGE ID}/{SCRIPT PATH AND FILE NAME (.js)}`:

- The `{PACKAGE ID}` placeholder is the library's [package ID](#). The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file.
- The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and file name under `wwwroot`.

HTML


```
<body>
    ...

    <script src="{BLAZOR SCRIPT}"></script>
    <script src="_content/{PACKAGE ID}/{SCRIPT PATH AND FILE NAME (.js)}">
</script>
</body>
```

In the following example of the preceding `<script>` tag:

- The Razor class library has an assembly name of `ComponentLibrary`, and a `<PackageId>` isn't specified in the library's project file.
- The `scripts.js` file is in the class library's `wwwroot` folder.

HTML

```
<script src="_content/ComponentLibrary/scripts.js"></script>
```

For more information, see [Consume ASP.NET Core Razor components from a Razor class library \(RCL\)](#).

Inject a script before or after Blazor starts

To ensure scripts load before or after Blazor starts, use a JavaScript initializer. For more information and examples, see [ASP.NET Core Blazor startup](#).

JavaScript isolation in JavaScript modules

Blazor enables JavaScript (JS) isolation in standard [JS modules](#) [↗](#) (ECMAScript [specification](#) [↗](#)).

JS isolation provides the following benefits:

- Imported JS no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JS.

In server-side scenarios, always trap [JSDisconnectedException](#) in case loss of Blazor's SignalR circuit prevents a JS interop call from disposing a module, which results in an unhandled exception. Blazor WebAssembly apps don't use a SignalR connection during JS interop, so there's no need to trap [JSDisconnectedException](#) in Blazor WebAssembly apps for module disposal.

For more information, see the following resources:

- [JavaScript isolation in JavaScript modules](#)
- [JavaScript interop calls without a circuit](#)

Call JavaScript functions from .NET methods in ASP.NET Core Blazor

Article • 10/18/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to invoke JavaScript (JS) functions from .NET.

For information on how to call .NET methods from JS, see [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#).

Invoke JS functions

[IJSRuntime](#) is registered by the Blazor framework. To call into JS from .NET, inject the [IJSRuntime](#) abstraction and call one of the following methods:

- [IJSRuntime.InvokeAsync](#)
- [JSRuntimeExtensions.InvokeAsync](#)
- [JSRuntimeExtensions.InvokeVoidAsync](#)

For the preceding .NET methods that invoke JS functions:

- The function identifier (`String`) is relative to the global scope (`window`). To call `window.someScope.someFunction`, the identifier is `someScope.someFunction`. There's no need to register the function before it's called.
- Pass any number of JSON-serializable arguments in `Object[]` to a JS function.
- The cancellation token (`CancellationToken`) propagates a notification that operations should be canceled.
- `TimeSpan` represents a time limit for a JS operation.
- The `TValue` return type must also be JSON serializable. `TValue` should match the .NET type that best maps to the JSON type returned.
- A [JS Promise](#) is returned for `InvokeAsync` methods. `InvokeAsync` unwraps the [Promise](#) and returns the value awaited by the [Promise](#).

For Blazor apps with prerendering enabled, which is the default for server-side apps, calling into JS isn't possible during prerendering. For more information, see the [Prerendering](#) section.

The following example is based on [TextDecoder](#), a JS-based decoder. The example demonstrates how to invoke a JS function from a C# method that offloads a requirement from developer code to an existing JS API. The JS function accepts a byte array from a C# method, decodes the array, and returns the text to the component for display.

HTML

```
<script>
    window.convertArray = (win1251Array) => {
        var win1251decoder = new TextDecoder('windows-1251');
        var bytes = new Uint8Array(win1251Array);
        var decodedArray = win1251decoder.decode(bytes);
        return decodedArray;
    };
</script>
```

ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

The following component:

- Invokes the `convertArray` JS function with `InvokeAsync` when selecting a button (`Convert Array`).
- After the JS function is called, the passed array is converted into a string. The string is returned to the component for display (`text`).

`CallJs1.razor`:

razor

```
@page "/call-js-1"
@inject IJSRuntime JS

<PageTitle>Call JS 1</PageTitle>

<h1>Call JS Example 1</h1>

<p>
    <button @onclick="ConvertArray">Convert Array</button>
```

```

</p>

<p>
    @text
</p>

<p>
    Quote @2005 <a href="https://www.uphe.com">Universal Pictures</a>:
    <a href="https://www.uphe.com/movies/serenity-2005">Serenity</a><br>
    <a href="https://www.imdb.com/name/nm0472710/">David Krumholtz on
    IMDB</a>
</p>

@code {
    private MarkupString text;

    private uint[] quoteArray =
        new uint[]
        {
            60, 101, 109, 62, 67, 97, 110, 39, 116, 32, 115, 116, 111, 112,
32,
            116, 104, 101, 32, 115, 105, 103, 110, 97, 108, 44, 32, 77, 97,
108, 46, 60, 47, 101, 109, 62, 32, 45, 32, 77, 114, 46, 32, 85,
110,
            105, 118, 101, 114, 115, 101, 10, 10,
        };

    private async Task ConvertArray() =>
    {
        text = new(await JS.InvokeAsync<string>("convertArray",
quoteArray));
    }
}

```

JavaScript API restricted to user gestures

This section applies to server-side components.

Some browser JavaScript (JS) APIs can only be executed in the context of a user gesture, such as using the [Fullscreen API \(MDN documentation\)](#). These APIs can't be called through the JS interop mechanism in server-side components because UI event handling is performed asynchronously and generally no longer in the context of the user gesture. The app must handle the UI event completely in JavaScript, so use `onclick` instead of Blazor's `@onlick` directive attribute.

Invoke JavaScript functions without reading a returned value (`InvokeVoidAsync`)

Use `InvokeVoidAsync` when:

- .NET isn't required to read the result of a JavaScript (JS) call.
- JS functions return `void(0)/void 0` or `undefined`.

Provide a `displayTickerAlert1` JS function. The function is called with `InvokeVoidAsync` and doesn't return a value:

HTML

```
<script>
  window.displayTickerAlert1 = (symbol, price) => {
    alert(`${symbol}: ${price}!`);
  };
</script>
```

ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

Component (`.razor`) example (`InvokeVoidAsync`)

`TickerChanged` calls the `handleTickerChanged1` method in the following component.

`CallJs2.razor`:

razor

```
@page "/call-js-2"
@inject IJSRuntime JS

<PageTitle>Call JS 2</PageTitle>

<h1>Call JS Example 2</h1>

<p>
  <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{
  <p>@stockSymbol price: @price.ToString("c")</p>
}

@code {
  private string? stockSymbol;
  private decimal price;
```

```

private async Task SetStock()
{
    stockSymbol =
        $"{(char)('A' + Random.Shared.Next(0, 26))}" +
        $"{(char)('A' + Random.Shared.Next(0, 26))}";
    price = Random.Shared.Next(1, 101);
    await JS.InvokeVoidAsync("displayTickerAlert1", stockSymbol, price);
}
}

```

Class (.cs) example (InvokeVoidAsync)

JsInteropClasses1.cs:

C#

```

using Microsoft.JSInterop;

namespace BlazorSample;

public class JsInteropClasses1(IJSRuntime js) : IDisposable
{
    private readonly IJSRuntime js = js;

    public async ValueTask TickerChanged(string symbol, decimal price) =>
        await js.InvokeVoidAsync("displayTickerAlert1", symbol, price);

    // Calling SuppressFinalize(this) prevents derived types that introduce
    // a finalizer from needing to re-implement IDisposable.
    public void Dispose() => GC.SuppressFinalize(this);
}

```

TickerChanged calls the handleTickerChanged1 method in the following component.

CallJs3.razor:

razor

```

@page "/call-js-3"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 3</PageTitle>

<h1>Call JS Example 3</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

```

```

@if (stockSymbol is not null)
{
    <p>@stockSymbol price: @price.ToString("c")</p>
}

@code {
    private string? stockSymbol;
    private decimal price;
    private JsInteropClasses1? jsClass;

    protected override void OnInitialized() => jsClass = new(JS);

    private async Task SetStock()
    {
        if (jsClass is not null)
        {
            stockSymbol =
                $"{(char)('A' + Random.Shared.Next(0, 26))}" +
                $"{(char)('A' + Random.Shared.Next(0, 26))}";
            price = Random.Shared.Next(1, 101);
            await jsClass.TickerChanged(stockSymbol, price);
        }
    }

    public void Dispose() => jsClass?.Dispose();
}

```

Invoke JavaScript functions and read a returned value (InvokeAsync)

Use [InvokeAsync](#) when .NET should read the result of a JavaScript (JS) call.

Provide a `displayTickerAlert2` JS function. The following example returns a string for display by the caller:

HTML

```

<script>
    window.displayTickerAlert2 = (symbol, price) => {
        if (price < 20) {
            alert(`${symbol}: ${price}!`);
            return "User alerted in the browser.";
        } else {
            return "User NOT alerted.";
        }
    };
</script>

```


ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

Component (`.razor`) example (`InvokeAsync`)

`TickerChanged` calls the `handleTickerChanged2` method and displays the returned string in the following component.

`CallJs4.razor`:

razor

```
@page "/call-js-4"
@inject IJSRuntime JS

<PageTitle>Call JS 4</PageTitle>

<h1>Call JS Example 4</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{
    <p>@stockSymbol price: @price.ToString("c")</p>
}

@if (result is not null)
{
    <p>@result</p>
}

@code {
    private string? stockSymbol;
    private decimal price;
    private string? result;

    private async Task SetStock()
    {
        stockSymbol =
            $"{(char)('A' + Random.Shared.Next(0, 26))}" +
            $"{(char)('A' + Random.Shared.Next(0, 26))}";
        price = Random.Shared.Next(1, 101);
        var interopResult =
            await JS.InvokeAsync<string>("displayTickerAlert2", stockSymbol,
price);
```

```

        result = $"Result of TickerChanged call for {stockSymbol} at " +
            $"{price.ToString("c")}: {interopResult}";
    }
}

```

Class (.cs) example (InvokeAsync)

JsInteropClasses2.cs:

C#

```

using Microsoft.JSInterop;

namespace BlazorSample;

public class JsInteropClasses2(IJSRuntime js) : IDisposable
{
    private readonly IJSRuntime js = js;

    public async ValueTask<string> TickerChanged(string symbol, decimal
price) =>
        await js.InvokeAsync<string>("displayTickerAlert2", symbol, price);

    // Calling SuppressFinalize(this) prevents derived types that introduce
    // a finalizer from needing to re-implement IDisposable.
    public void Dispose() => GC.SuppressFinalize(this);
}

```

TickerChanged calls the handleTickerChanged2 method and displays the returned string in the following component.

CallJs5.razor:

razor

```

@page "/call-js-5"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 5</PageTitle>

<h1>Call JS Example 5</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{

```

```

        <p>@stockSymbol price: @price.ToString("c")</p>
    }

    @if (result is not null)
    {
        <p>@result</p>
    }

    @code {
        private string? stockSymbol;
        private decimal price;
        private JsInteropClasses2? jsClass;
        private string? result;

        protected override void OnInitialized() => jsClass = new(JS);

        private async Task SetStock()
        {
            if (jsClass is not null)
            {
                stockSymbol =
                    $"{(char)('A' + Random.Shared.Next(0, 26))}" +
                    $"{(char)('A' + Random.Shared.Next(0, 26))}";
                price = Random.Shared.Next(1, 101);
                var interopResult = await jsClass.TickerChanged(stockSymbol,
price);
                result = $"Result of TickerChanged call for {stockSymbol} at " +
                    $"{price.ToString("c")}: {interopResult}";
            }
        }

        public void Dispose() => jsClass?.Dispose();
    }

```

Dynamic content generation scenarios

For dynamic content generation with [BuildRenderTree](#), use the `[Inject]` attribute:

```
razor
```

```
[Inject]
IJSRuntime JS { get; set; }
```

Prerendering

This section applies to server-side apps that prerender Razor components. Prerendering is covered in [Prerender ASP.NET Core Razor components](#).

❗ Note

Internal navigation for [interactive routing](#) in Blazor Web Apps doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests. If the app adopts interactive routing, perform a full page reload for component examples that demonstrate prerendering behavior. For more information, see [Prerender ASP.NET Core Razor components](#).

During prerendering, calling into JavaScript (JS) isn't possible. The following example demonstrates how to use JS interop as part of a component's initialization logic in a way that's compatible with prerendering.

The following `scrollElementIntoView` function:

- Scrolls to the passed element with [scrollIntoView](#) [↗].
- Returns the element's `top` property value from the [getBoundingClientRect](#) [↗] method.

JavaScript

```
window.scrollElementIntoView = (element) => {  
    element.scrollIntoView();  
    return element.getBoundingClientRect().top;  
}
```

Where `IJSRuntime.InvokeAsync` calls the JS function in component code, the `ElementReference` is only used in `OnAfterRenderAsync` and not in any earlier lifecycle method because there's no HTML DOM element until after the component is rendered.

`StateHasChanged` ([reference source](#)) is called to enqueue rerendering of the component with the new state obtained from the JS interop call (for more information, see [ASP.NET Core Razor component rendering](#)). An infinite loop isn't created because `StateHasChanged` is only called when `scrollTop` is `null`.

PrerenderedInterop.razor:

razor

```
@page "/prerendered-interop"  
@using Microsoft.AspNetCore.Components  
@using Microsoft.JSInterop  
@inject IJSRuntime JS  
  
<PageTitle>Prerendered Interop</PageTitle>
```

```

<h1>Prerendered Interop Example</h1>

<div @ref="divElement" style="margin-top:2000px">
    Set value via JS interop call: <strong>@scrollTop</strong>
</div>

@code {
    private ElementReference divElement;
    private double? scrollTop;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender && scrollTop is null)
        {
            scrollTop = await JS.InvokeAsync<double>(
                "scrollElementIntoView", divElement);

            StateHasChanged();
        }
    }
}

```

The preceding example pollutes the client with a global function. For a better approach in production apps, see [JavaScript isolation in JavaScript modules](#).

Synchronous JS interop in client-side components

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from .NET to JavaScript in a client-side component, cast [IJSRuntime](#) to [IJSInProcessRuntime](#) to make the JS interop call:

```
razor
```

```
@inject IJSRuntime JS
```

```

...

@code {
    protected override void HandleSomeEvent()
    {
        var jsInProcess = (IJSInProcessRuntime)JS;
        var value = jsInProcess.Invoke<string>
("javascriptFunctionIdentifier");
    }
}

```

When working with [IJSObjectReference](#) in ASP.NET Core 5.0 or later client-side components, you can use [IJSInProcessObjectReference](#) synchronously instead. [IJSInProcessObjectReference](#) implements [IAsyncDisposable](#)/[IDisposable](#) and should be disposed for garbage collection to prevent a memory leak, as the following example demonstrates:

```

razor

@inject IJSRuntime JS
@implements IAsyncDisposable

...

@code {
    ...
    private IJSInProcessObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSInProcessObjectReference>
("import",
    "./scripts.js");
        }
    }

    ...

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}

```

JavaScript location

Load JavaScript (JS) code using any of approaches described by the [article on JavaScript location](#):

- Load a script in `<head>` markup (*Not generally recommended*)
- Load a script in `<body>` markup
- Load a script from an external JavaScript file (.js) collocated with a component
- Load a script from an external JavaScript file (.js)
- Inject a script before or after Blazor starts

For information on isolating scripts in [JS modules](#), see the [JavaScript isolation in JavaScript modules](#) section.

Warning

Only place a `<script>` tag in a component file (`.razor`) if the component is guaranteed to adopt static server-side rendering (static SSR) because the `<script>` tag can't be updated dynamically.

JavaScript isolation in JavaScript modules

Blazor enables JavaScript (JS) isolation in standard [JavaScript modules](#) (ECMAScript [specification](#)). JavaScript module loading works the same way in Blazor as it does for other types of web apps, and you're free to customize how modules are defined in your app. For a guide on how to use JavaScript modules, see [MDN Web Docs: JavaScript modules](#).

JS isolation provides the following benefits:

- Imported JS no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JS.

[Dynamic import with the `import\(\)` operator](#) is supported with ASP.NET Core and Blazor:

JavaScript

```
if ({CONDITION}) import("/additionalModule.js");
```

In the preceding example, the `{CONDITION}` placeholder represents a conditional check to determine if the module should be loaded.

For browser compatibility, see [Can I use: JavaScript modules: dynamic import](#).

For example, the following JS module exports a JS function for showing a [browser window prompt](#). Place the following JS code in an external JS file.

`wwwroot/scripts.js:`

JavaScript

```
export function showPrompt(message) {  
    return prompt(message, 'Type anything here');  
}
```

Add the preceding JS module to an app or class library as a static web asset in the `wwwroot` folder and then import the module into the .NET code by calling [InvokeAsync](#) on the [IJSRuntime](#) instance.

[IJSRuntime](#) imports the module as an [IJSObjectReference](#), which represents a reference to a JS object from .NET code. Use the [IJSObjectReference](#) to invoke exported JS functions from the module.

`CallJs6.razor:`

razor

```
@page "/call-js-6"  
@implements IAsyncDisposable  
@inject IJSRuntime JS  
  
<PageTitle>Call JS 6</PageTitle>  
  
<h1>Call JS Example 6</h1>  
  
<p>  
    <button @onclick="TriggerPrompt">Trigger browser window prompt</button>  
</p>  
  
<p>  
    @result  
</p>  
  
@code {  
    private IJSObjectReference? module;  
    private string? result;  
  
    protected override async Task OnAfterRenderAsync(bool firstRender)  
    {  
        if (firstRender)  
        {  
            module = await JS.InvokeAsync<IJSObjectReference>("import",
```



```

        "./scripts.js");
    }
}

private async Task TriggerPrompt() => result = await Prompt("Provide
text");

public async ValueTask<string?> Prompt(string message) =>
    module is not null ?
        await module.InvokeAsync<string>("showPrompt", message) : null;

async ValueTask IAsyncDisposable.DisposeAsync()
{
    if (module is not null)
    {
        await module.DisposeAsync();
    }
}
}

```

In the preceding example:

- By convention, the `import` identifier is a special identifier used specifically for importing a JS module.
- Specify the module's external JS file using its stable static web asset path: `./{SCRIPT PATH AND FILE NAME (.js)}`, where:
 - The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.
 - The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and file name under `wwwroot`.
- Disposes the `IJSObjectReference` for garbage collection in `IAsyncDisposable.DisposeAsync`.

Dynamically importing a module requires a network request, so it can only be achieved asynchronously by calling `InvokeAsync`.

`IJSInProcessObjectReference` represents a reference to a JS object whose functions can be invoked synchronously in client-side components. For more information, see the [Synchronous JS interop in client-side components](#) section.

ⓘ Note

When the external JS file is supplied by a [Razor class library](#), specify the module's JS file using its stable static web asset path: `./_content/{PACKAGE ID}/{SCRIPT PATH AND FILE NAME (.js)}`:

- The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.
- The `{PACKAGE ID}` placeholder is the library's [package ID](#). The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file. In the following example, the library's assembly name is `ComponentLibrary` and the library's project file doesn't specify `<PackageId>`.
- The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and file name under `wwwroot`. In the following example, the external JS file (`script.js`) is placed in the class library's `wwwroot` folder.
- `module` is a private nullable [IJSObjectReference](#) of the component class (`private IJSObjectReference? module;`).

C#

```
module = await js.InvokeAsync<IJSObjectReference>(
    "import", "./_content/ComponentLibrary/scripts.js");
```

For more information, see [Consume ASP.NET Core Razor components from a Razor class library \(RCL\)](#).

Throughout the Blazor documentation, examples use the `.js` file extension for module files, not the [newer .mjs file extension \(RFC 9239\)](#). Our documentation continues to use the `.js` file extension for the same reasons the Mozilla Foundation's documentation continues to use the `.js` file extension. For more information, see [Aside — .mjs versus .js \(MDN documentation\)](#).

Capture references to elements

Some JavaScript (JS) interop scenarios require references to HTML elements. For example, a UI library may require an element reference for initialization, or you might need to call command-like APIs on an element, such as `click` or `play`.

Capture references to HTML elements in a component using the following approach:

- Add an `@ref` attribute to the HTML element.
- Define a field of type [ElementReference](#) whose name matches the value of the `@ref` attribute.

The following example shows capturing a reference to the `username` `<input>` element:

razor

```
<input @ref="username" ... />

@code {
    private ElementReference username;
}
```

⚠ Warning

Only use an element reference to mutate the contents of an empty element that doesn't interact with Blazor. This scenario is useful when a third-party API supplies content to the element. Because Blazor doesn't interact with the element, there's no possibility of a conflict between Blazor's representation of the element and the DOM.

In the following example, it's *dangerous* to mutate the contents of the unordered list (`ul`) using `MyList` via JS interop because Blazor interacts with the DOM to populate this element's list items (``) from the `Todos` object:

razor

```
<ul @ref="MyList">
    @foreach (var item in Todos)
    {
        <li>@item.Text</li>
    }
</ul>
```

Using the `MyList` element reference to merely read DOM content or trigger an event is supported.

If JS interop *mutates the contents* of element `MyList` and Blazor attempts to apply diffs to the element, the diffs won't match the DOM. Modifying the contents of the list via JS interop with the `MyList` element reference is ***not supported***.

For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

An `ElementReference` is passed through to JS code via JS interop. The JS code receives an `HTMLElement` instance, which it can use with normal DOM APIs. For example, the following code defines a .NET extension method (`TriggerClickEvent`) that enables sending a mouse click to an element.

The JS function `clickElement` creates a [click](#) event on the passed HTML element (`element`):

JavaScript

```
window.interopFunctions = {
    clickElement : function (element) {
        element.click();
    }
}
```

To call a JS function that doesn't return a value, use [JSRuntimeExtensions.InvokeVoidAsync](#). The following code triggers a client-side [click](#) event by calling the preceding JS function with the captured [ElementReference](#):

razor

```
@inject IJSRuntime JS

<button @ref="exampleButton">Example Button</button>

<button @onclick="TriggerClick">
    Trigger click event on <code>Example Button</code>
</button>

@code {
    private ElementReference exampleButton;

    public async Task TriggerClick()
    {
        await JS.InvokeVoidAsync(
            "interopFunctions.clickElement", exampleButton);
    }
}
```

To use an extension method, create a static extension method that receives the [IJSRuntime](#) instance:

C#

```
public static async Task TriggerClickEvent(this ElementReference elementRef,
    IJSRuntime js)
{
    await js.InvokeVoidAsync("interopFunctions.clickElement", elementRef);
}
```

The `clickElement` method is called directly on the object. The following example assumes that the `TriggerClickEvent` method is available from the `JsInteropClasses`

namespace:

razor

```
@inject IJSRuntime JS
@using JsInteropClasses

<button @ref="exampleButton">Example Button</button>

<button @onclick="TriggerClick">
    Trigger click event on <code>Example Button</code>
</button>

@code {
    private ElementReference exampleButton;

    public async Task TriggerClick()
    {
        await exampleButton.TriggerClickEvent(JS);
    }
}
```

Important

The `exampleButton` variable is only populated after the component is rendered. If an unpopulated [ElementReference](#) is passed to JS code, the JS code receives a value of `null`. To manipulate element references after the component has finished rendering, use the [OnAfterRenderAsync or OnAfterRender component lifecycle methods](#).

When working with generic types and returning a value, use [ValueTask<TResult>](#):

C#

```
public static ValueTask<T> GenericMethod<T>(
    this ElementReference elementRef, IJSRuntime js) =>
    js.InvokeAsync<T>("{JAVASCRIPT FUNCTION}", elementRef);
```

The `{JAVASCRIPT FUNCTION}` placeholder is the JS function identifier.

`GenericMethod` is called directly on the object with a type. The following example assumes that the `GenericMethod` is available from the `JsInteropClasses` namespace:

razor

```

@Inject IJSRuntime JS
@using JsInteropClasses

<input @ref="username" />

<button @onclick="OnClickMethod">Do something generic</button>

<p>
    returnValue: @returnValue
</p>

@code {
    private ElementReference username;
    private string? returnValue;

    private async Task OnClickMethod()
    {
        returnValue = await username.GenericMethod<string>(JS);
    }
}

```

Reference elements across components

An [ElementReference](#) can't be passed between components because:

- The instance is only guaranteed to exist after the component is rendered, which is during or after a component's [OnAfterRender/OnAfterRenderAsync](#) method executes.
- An [ElementReference](#) is a [struct](#), which can't be passed as a [component parameter](#).

For a parent component to make an element reference available to other components, the parent component can:

- Allow child components to register callbacks.
- Invoke the registered callbacks during the [OnAfterRender](#) event with the passed element reference. Indirectly, this approach allows child components to interact with the parent's element reference.

HTML

```

<style>
    .red { color: red }
</style>

```

HTML

```
<script>
    function setElementClass(element, className) {
        var myElement = element;
        myElement.classList.add(className);
    }
</script>
```

ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

CallJs7.razor (parent component):

razor

```
@page "/call-js-7"

<PageTitle>Call JS 7</PageTitle>

<h1>Call JS Example 7</h1>

<h2 @ref="title">Hello, world!</h2>

Welcome to your new app.

<SurveyPrompt Parent="this" Title="How is Blazor working for you?" />
```

CallJs7.razor.cs:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.Pages;

public partial class CallJs7 :
    ComponentBase, IObservable<ElementReference>, IDisposable
{
    private bool disposing;
    private readonly List<IObserver<ElementReference>> subscriptions = [];
    private ElementReference title;

    protected override void OnAfterRender(bool firstRender)
    {
        base.OnAfterRender(firstRender);

        foreach (var subscription in subscriptions)
```

```

        {
            subscription.OnNext(title);
        }
    }

    public void Dispose()
    {
        disposing = true;

        foreach (var subscription in subscriptions)
        {
            try
            {
                subscription.OnCompleted();
            }
            catch (Exception)
            {
            }
        }

        subscriptions.Clear();

        // The following prevents derived types that introduce a
        // finalizer from needing to re-implement IDisposable.
        GC.SuppressFinalize(this);
    }

    public IDisposable Subscribe(IObserver<ElementReference> observer)
    {
        if (disposing)
        {
            throw new InvalidOperationException("Parent being disposed");
        }

        subscriptions.Add(observer);

        return new Subscription(observer, this);
    }

    private class Subscription(IObserver<ElementReference> observer,
        CallJs7 self) : IDisposable
    {
        public IObservable<ElementReference> Observer { get; } = observer;
        public CallJs7 Self { get; } = self;

        public void Dispose() => Self.subscriptions.Remove(Observer);
    }
}

```

In the preceding example, the namespace of the app is `BlazorSample`. If testing the code locally, update the namespace.

`SurveyPrompt.razor` (child component):

razor

```
<div class="alert alert-secondary mt-4">
    <span class="oi oi-pencil me-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold link-dark"
href="https://go.microsoft.com/fwlink/?linkid=2186158">brief survey</a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string? Title { get; set; }
}
```

SurveyPrompt.razor.cs:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.JSInterop;

namespace BlazorSample.Components;

public partial class SurveyPrompt :
    ComponentBase, IObservable<ElementReference>, IDisposable
{
    private IDisposable? subscription = null;

    [Parameter]
    public IObservable<ElementReference>? Parent { get; set; }

    [Inject]
    public IJSRuntime? JS {get; set;}

    protected override void OnParametersSet()
    {
        base.OnParametersSet();

        subscription?.Dispose();
        subscription = Parent?.Subscribe(this);
    }

    public void OnCompleted() => subscription = null;

    public void OnError(Exception error) => subscription = null;

    public void OnNext(ElementReference value) =>
```

```

        _ = (JS?.InvokeAsync<object>("setElementClass", [value, "red"]));

    public void Dispose()
    {
        subscription?.Dispose();

        // The following prevents derived types that introduce a
        // finalizer from needing to re-implement IDisposable.
        GC.SuppressFinalize(this);
    }
}

```

In the preceding example, the namespace of the app is `BlazorSample` with shared components in the `Shared` folder. If testing the code locally, update the namespace.

Harden JavaScript interop calls

This section only applies to Interactive Server components, but client-side components may also set JS interop timeouts if conditions warrant it.

In server-side apps with server interactivity, JavaScript (JS) interop may fail due to networking errors and should be treated as unreliable. Blazor apps use a one minute timeout for JS interop calls. If an app can tolerate a more aggressive timeout, set the timeout using one of the following approaches.

Set a global timeout in the `Program.cs` with `CircuitOptions.JSInteropDefaultCallTimeout`:

C#

```

builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents(options =>
        options.JSInteropDefaultCallTimeout = {TIMEOUT});

```

The `{TIMEOUT}` placeholder is a [TimeSpan](#) (for example, `TimeSpan.FromSeconds(80)`).

Set a per-invocation timeout in component code. The specified timeout overrides the global timeout set by `JSInteropDefaultCallTimeout`:

C#

```

var result = await JS.InvokeAsync<string>("{ID}", {TIMEOUT}, new[] { "Arg1"
});

```

In the preceding example:

- The `{TIMEOUT}` placeholder is a [TimeSpan](#) (for example, `TimeSpan.FromSeconds(80)`).
- The `{ID}` placeholder is the identifier for the function to invoke. For example, the value `someScope.someFunction` invokes the function `window.someScope.someFunction`.

Although a common cause of JS interop failures are network failures with server-side components, per-invocation timeouts can be set for JS interop calls for client-side components. Although no SignalR circuit exists for a client-side component, JS interop calls might fail for other reasons that apply.

For more information on resource exhaustion, see [Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering](#).

Avoid circular object references

Objects that contain circular references can't be serialized on the client for either:

- .NET method calls.
- JavaScript method calls from C# when the return type has circular references.

JavaScript libraries that render UI

Sometimes you may wish to use JavaScript (JS) libraries that produce visible user interface elements within the browser DOM. At first glance, this might seem difficult because Blazor's diffing system relies on having control over the tree of DOM elements and runs into errors if some external code mutates the DOM tree and invalidates its mechanism for applying diffs. This isn't a Blazor-specific limitation. The same challenge occurs with any diff-based UI framework.

Fortunately, it's straightforward to embed externally-generated UI within a Razor component UI reliably. The recommended technique is to have the component's code (`.razor` file) produce an empty element. As far as Blazor's diffing system is concerned, the element is always empty, so the renderer does not recurse into the element and instead leaves its contents alone. This makes it safe to populate the element with arbitrary externally-managed content.

The following example demonstrates the concept. Within the `if` statement when `firstRender` is `true`, interact with `unmanagedElement` outside of Blazor using JS interop. For example, call an external JS library to populate the element. Blazor leaves the element's contents alone until this component is removed. When the component is removed, the component's entire DOM subtree is also removed.

razor

```
<h1>Hello! This is a Razor component rendered at @DateTime.Now</h1>

<div @ref="unmanagedElement"></div>

@code {
    private ElementReference unmanagedElement;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            ...
        }
    }
}
```

Consider the following example that renders an interactive map using [open-source Mapbox APIs](#).

The following JS module is placed into the app or made available from a Razor class library.

ⓘ Note

To create the [Mapbox](#) map, obtain an access token from [Mapbox Sign in](#) and provide it where the `{ACCESS TOKEN}` appears in the following code.

wwwroot/mapComponent.js:

JavaScript

```
import 'https://api.mapbox.com/mapbox-gl-js/v1.12.0/mapbox-gl.js';

mapboxgl.accessToken = '{ACCESS TOKEN}';

export function addMapToElement(element) {
    return new mapboxgl.Map({
        container: element,
        style: 'mapbox://styles/mapbox/streets-v11',
        center: [-74.5, 40],
        zoom: 9
    });
}

export function setMapCenter(map, latitude, longitude) {
```

```
map.setCenter([longitude, latitude]);
}
```

To produce correct styling, add the following stylesheet tag to the host HTML page.

Add the following `<link>` element to the `<head>` element markup (location of `<head>` content):

HTML

```
<link href="https://api.mapbox.com/mapbox-gl-js/v1.12.0/mapbox-gl.css"
      rel="stylesheet" />
```

CallJs8.razor:

razor

```
@page "/call-js-8"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 8</PageTitle>

<HeadContent>
    <link href="https://api.mapbox.com/mapbox-gl-js/v1.12.0/mapbox-gl.css"
          rel="stylesheet" />
</HeadContent>

<h1>Call JS Example 8</h1>

<div @ref="mapElement" style='width:400px;height:300px'></div>

<button @onclick="() => ShowAsync(51.454514, -2.587910)">Show Bristol,
UK</button>
<button @onclick="() => ShowAsync(35.6762, 139.6503)">Show Tokyo,
Japan</button>

@code
{
    private ElementReference mapElement;
    private IJSObjectReference? mapModule;
    private IJSObjectReference? mapInstance;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            mapModule = await JS.InvokeAsync<IJSObjectReference>(
                "import", "./mapComponent.js");
            mapInstance = await mapModule.InvokeAsync<IJSObjectReference>(
                "addMapToElement", mapElement);
        }
    }
}
```

```

    }
}

private async Task ShowAsync(double latitude, double longitude)
{
    if (mapModule is not null && mapInstance is not null)
    {
        await mapModule.InvokeVoidAsync("setMapCenter", mapInstance,
            latitude, longitude);
    }
}

async ValueTask IAsyncDisposable.DisposeAsync()
{
    if (mapInstance is not null)
    {
        await mapInstance.DisposeAsync();
    }

    if (mapModule is not null)
    {
        await mapModule.DisposeAsync();
    }
}
}

```

The preceding example produces an interactive map UI. The user:

- Can drag to scroll or zoom.
- Select buttons to jump to predefined locations.



In the preceding example:

- The `<div>` with `@ref="mapElement"` is left empty as far as Blazor is concerned. The `mapbox-gl.js` script can safely populate the element and modify its contents. Use this technique with any JS library that renders UI. You can embed components from a third-party JS SPA framework inside Razor components, as long as they don't try to reach out and modify other parts of the page. It is **not** safe for external JS code to modify elements that Blazor does not regard as empty.
- When using this approach, bear in mind the rules about how Blazor retains or destroys DOM elements. The component safely handles button click events and updates the existing map instance because DOM elements are retained where possible. If you were rendering a list of map elements from inside a `@foreach` loop, you want to use `@key` to ensure the preservation of component instances. Otherwise, changes in the list data could cause component instances to retain the state of previous instances in an undesirable manner. For more information, see [how to use the @key directive attribute to preserve the relationship among elements, components, and model objects](#).
- The example encapsulates JS logic and dependencies within a JavaScript module and loads the module dynamically using the `import` identifier. For more information, see [JavaScript isolation in JavaScript modules](#).

Byte array support

Blazor supports optimized byte array JavaScript (JS) interop that avoids encoding/decoding byte arrays into Base64. The following example uses JS interop to pass a byte array to JavaScript.

Provide a `receiveByteArray` JS function. The function is called with `InvokeVoidAsync` and doesn't return a value:

HTML

```
<script>
  window.receiveByteArray = (bytes) => {
    let utf8decoder = new TextDecoder();
    let str = utf8decoder.decode(bytes);
    return str;
  };
</script>
```

ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

`CallJs9.razor`:

razor

```
@page "/call-js-9"
@inject IJSRuntime JS

<h1>Call JS Example 9</h1>

<p>
  <button @onclick="SendByteArray">Send Bytes</button>
</p>

<p>
  @result
</p>

<p>
  Quote &copy;2005 <a href="https://www.uphe.com">Universal Pictures</a>:
  <a href="https://www.uphe.com/movies/serenity-2005">Serenity</a><br>
  <a href="https://www.imdb.com/name/nm0821612/">Jewel Staite on IMDB</a>
</p>

@code {
```



```

private string? result;

private async Task SendByteArray()
{
    var bytes = new byte[] { 0x45, 0x76, 0x65, 0x72, 0x79, 0x74, 0x68,
0x69,
        0x6e, 0x67, 0x27, 0x73, 0x20, 0x73, 0x68, 0x69, 0x6e, 0x79,
0x2c,
        0x20, 0x43, 0x61, 0x70, 0x74, 0x69, 0x61, 0x6e, 0x2e, 0x20,
0x4e,
        0x6f, 0x74, 0x20, 0x74, 0x6f, 0x20, 0x66, 0x72, 0x65, 0x74, 0x2e
    };

    result = await JS.InvokeAsync<string>("receiveByteArray", bytes);
}
}

```

Stream from .NET to JavaScript

Blazor supports streaming data directly from .NET to JavaScript (JS). Streams are created using a [DotNetStreamReference](#).

[DotNetStreamReference](#) represents a .NET stream and uses the following parameters:

- `stream`: The stream sent to JS.
- `leaveOpen`: Determines if the stream is left open after transmission. If a value isn't provided, `leaveOpen` defaults to `false`.

In JS, use an array buffer or a readable stream to receive the data:

- Using an [ArrayBuffer](#) ↗:

JavaScript

```

async function streamToJavaScript(streamRef) {
    const data = await streamRef.arrayBuffer();
}

```

- Using a [ReadableStream](#) ↗:

JavaScript

```

async function streamToJavaScript(streamRef) {
    const stream = await streamRef.stream();
}

```

In C# code:

C#

```
var streamRef = new DotNetStreamReference(stream: {STREAM}, leaveOpen:
false);
await JS.InvokeVoidAsync("streamToJavaScript", streamRef);
```

In the preceding example:

- The `{STREAM}` placeholder represents the [Stream](#) sent to JS.
- `JS` is an injected [IJSRuntime](#) instance.

Disposing a [DotNetStreamReference](#) instance is usually unnecessary. When `leaveOpen` is set to its default value of `false`, the underlying [Stream](#) is automatically disposed after transmission to JS.

If `leaveOpen` is `true`, then disposing a [DotNetStreamReference](#) doesn't dispose its underlying [Stream](#). The app's code determines when to dispose the underlying [Stream](#). When deciding how to dispose the underlying [Stream](#), consider the following:

- Disposing a [Stream](#) while it's being transmitted to JS is considered an application error and may cause an unhandled exception to occur.
- [Stream](#) transmission begins as soon as the [DotNetStreamReference](#) is passed as an argument to a JS interop call, regardless of whether the stream is actually used in JS logic.

Given these characteristics, we recommend disposing the underlying [Stream](#) only after it's fully consumed by JS (the promise returned by `arrayBuffer` or `stream` resolves). It follows that a [DotNetStreamReference](#) should only be passed to JS if it's unconditionally going to be consumed by JS logic.

[Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#) covers the reverse operation, streaming from JavaScript to .NET.

[ASP.NET Core Blazor file downloads](#) covers how to download a file in Blazor.

Catch JavaScript exceptions

To catch JS exceptions, wrap the JS interop in a [try-catch block](#) and catch a [JSException](#).

In the following example, the `nonFunction` JS function doesn't exist. When the function isn't found, the [JSException](#) is trapped with a [Message](#) that indicates the following error:

```
Could not find 'nonFunction' ('nonFunction' was undefined).
```

CallJs11.razor:

razor

```
@page "/call-js-11"
@inject IJSRuntime JS

<PageTitle>Call JS 11</PageTitle>

<h1>Call JS Example 11</h1>

<p>
    <button @onclick="CatchUndefinedJSFunction">Catch Exception</button>
</p>

<p>
    @result
</p>

<p>
    @errorMessage
</p>

@code {
    private string? errorMessage;
    private string? result;

    private async Task CatchUndefinedJSFunction()
    {
        try
        {
            result = await JS.InvokeAsync<string>("nonFunction");
        }
        catch (JSException e)
        {
            errorMessage = $"Error Message: {e.Message}";
        }
    }
}
```

Abort a long-running JavaScript function

Use a JS [AbortController](#) with a [CancellationTokenSource](#) in the component to abort a long-running JavaScript function from C# code.

The following JS `Helpers` class contains a simulated long-running function, `LongRunningFn`, to count continuously until the [AbortController.signal](#) indicates that [AbortController.abort](#) has been called. The `sleep` function is for demonstration purposes to simulate slow execution of the long-running function and wouldn't be

present in production code. When a component calls `stopFn`, the `longRunningFn` is signalled to abort via the `while` loop conditional check on `AbortSignal.aborted` [↗](#).

HTML

```
<script>
  class Helpers {
    static #controller = new AbortController();

    static async #sleep(ms) {
      return new Promise(resolve => setTimeout(resolve, ms));
    }

    static async longRunningFn() {
      var i = 0;
      while (!this.#controller.signal.aborted) {
        i++;
        console.log(`longRunningFn: ${i}`);
        await this.#sleep(1000);
      }
    }

    static stopFn() {
      this.#controller.abort();
      console.log('longRunningFn aborted!');
    }
  }

  window.Helpers = Helpers;
</script>
```

ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

The following component:

- Invokes the JS function `longRunningFn` when the `Start Task` button is selected. A `CancellationTokenSource` is used to manage the execution of the long-running function. `CancellationToken.Register` sets a JS interop call delegate to execute the JS function `stopFn` when the `CancellationTokenSource.Token` is cancelled.
- When the `Cancel Task` button is selected, the `CancellationTokenSource.Token` is cancelled with a call to `Cancel`.
- The `CancellationTokenSource` is disposed in the `Dispose` method.

CallJs12.razor:

razor

```
@page "/call-js-12"
@inject IJSRuntime JS

<h1>Cancel long-running JS interop</h1>

<p>
    <button @onclick="StartTask">Start Task</button>
    <button @onclick="CancelTask">Cancel Task</button>
</p>

@code {
    private CancellationTokensource? cts;

    private async Task StartTask()
    {
        cts = new CancellationTokensource();
        cts.Token.Register(() => JS.InvokeVoidAsync("Helpers.stopFn"));

        await JS.InvokeVoidAsync("Helpers.longRunningFn");
    }

    private void CancelTask()
    {
        cts?.Cancel();
    }

    public void Dispose()
    {
        cts?.Cancel();
        cts?.Dispose();
    }
}
```

A browser's [developer tools](#) console indicates the execution of the long-running JS function after the **Start Task** button is selected and when the function is aborted after the **Cancel Task** button is selected:

Console

```
longRunningFn: 1
longRunningFn: 2
longRunningFn: 3
longRunningFn aborted!
```

JavaScript [JSImport] / [JSExport] interop

This section applies to client-side components.

As an alternative to interacting with JavaScript (JS) in client-side components using Blazor's JS interop mechanism based on the [IJSRuntime](#) interface, a JS `[JSImport]` / `[JSExport]` interop API is available to apps targeting .NET 7 or later.

For more information, see [JavaScript JSImport/JSExport interop with ASP.NET Core Blazor](#).

Unmarshalled JavaScript interop

This section applies to client-side components.

Unmarshalled interop using the [IJSUnmarshalledRuntime](#) interface is obsolete and should be replaced with JavaScript `[JSImport]` / `[JSExport]` interop.

For more information, see [JavaScript JSImport/JSExport interop with ASP.NET Core Blazor](#).

Disposal of JavaScript interop object references

Examples throughout the JavaScript (JS) interop articles demonstrate typical object disposal patterns:

- When calling JS from .NET, as described in this article, dispose any created [IJSObjectReference/IJSInProcessObjectReference](#) / `JSObjectReference` either from .NET or from JS to avoid leaking JS memory.
- When calling .NET from JS, as described in [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#), dispose of a created `DotNetObjectReference` either from .NET or from JS to avoid leaking .NET memory.

JS interop object references are implemented as a map keyed by an identifier on the side of the JS interop call that creates the reference. When object disposal is initiated from either the .NET or JS side, Blazor removes the entry from the map, and the object can be garbage collected as long as no other strong reference to the object is present.

At a minimum, always dispose objects created on the .NET side to avoid leaking .NET managed memory.

DOM cleanup tasks during component disposal

For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

JavaScript interop calls without a circuit

For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

Additional resources

- [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#)
- [InteropComponent.razor example \(dotnet/AspNetCore GitHub repository main branch\)](#) [↗]: The `main` branch represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release (for example, `release/{VERSION}`, where the `{VERSION}` placeholder is the release version), use the **Switch branches or tags** dropdown list to select the branch. For a branch that no longer exists, use the **Tags** tab to find the API (for example, `v7.0.0`).
- [Blazor samples GitHub repository \(dotnet/blazor-samples\)](#) [↗] ([how to download](#))
- [Handle errors in ASP.NET Core Blazor apps](#) (*JavaScript interop* section)
- [Threat mitigation: JavaScript functions invoked from .NET](#)

Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Article • 10/25/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to invoke .NET methods from JavaScript (JS).

For information on how to call JS functions from .NET, see [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#).

Invoke a static .NET method

To invoke a static .NET method from JavaScript (JS), use the JS functions:

- `DotNet.invokeMethodAsync` (*recommended*): Asynchronous for both server-side and client-side components.
- `DotNet.invokeMethod`: Synchronous for client-side components only.

Pass in the name of the assembly containing the method, the identifier of the static .NET method, and any arguments.

In the following example:

- The `{ASSEMBLY NAME}` placeholder is the app's assembly name.
- The `{.NET METHOD ID}` placeholder is the .NET method identifier.
- The `{ARGUMENTS}` placeholder are optional, comma-separated arguments to pass to the method, each of which must be JSON-serializable.

JavaScript

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', '{.NET METHOD ID}',  
{ARGUMENTS});
```


`DotNet.invokeMethodAsync` returns a [JS Promise](#) representing the result of the operation. `DotNet.invokeMethod` (client-side components) returns the result of the operation.

📘 Important

For server-side components, we recommend the asynchronous function (`invokeMethodAsync`) over the synchronous version (`invokeMethod`).

The .NET method must be public, static, and have the [\[JSInvokable\]](#) attribute.

In the following example:

- The `{<T>}` placeholder indicates the return type, which is only required for methods that return a value.
- The `{.NET METHOD ID}` placeholder is the method identifier.

razor

```
@code {
    [JSInvokable]
    public static Task<T> {.NET METHOD ID}()
    {
        ...
    }
}
```

⚠️ Note

Calling open generic methods isn't supported with static .NET methods but is supported with instance methods. For more information, see the [Call .NET generic class methods](#) section.

In the following component, the `ReturnArrayAsync` C# method returns an `int` array. The [\[JSInvokable\]](#) attribute is applied to the method, which makes the method invocable by JS.

`CallDotnet1.razor`:

razor

```
@page "/call-dotnet-1"
@implements IAsyncDisposable
```

```

@Inject IJSRuntime JS

<PageTitle>Call .NET 1</PageTitle>

<h1>Call .NET Example 1</h1>

<p>
    <button id="btn">Trigger .NET static method</button>
</p>

<p>
    See the result in the developer tools console.
</p>

@code {
    private IJSObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/Pages/CallDotnet1.razor.js");

            await module.InvokeVoidAsync("addHandlers");
        }
    }

    [JSInvokable]
    public static Task<int[]> ReturnArrayAsync() =>
        Task.FromResult(new int[] { 11, 12, 13 });

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}

```

CallDotnet1.razor.js:

JavaScript

```

export function returnArrayAsync() {
    DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync')
        .then(data => {
            console.log(data);
        });
}

export function addHandlers() {

```

```
const btn = document.getElementById("btn");
btn.addEventListener("click", returnArrayAsync);
}
```

The `addHandlers` JS function adds a [click](#) event to the button. The `returnArrayAsync` JS function is assigned as the handler.

The `returnArrayAsync` JS function calls the `ReturnArrayAsync` .NET method of the component, which logs the result to the browser's web developer tools console.

`BlazorSample` is the app's assembly name.

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

When the `Trigger .NET static method` button is selected, the browser's developer tools console output displays the array data. The format of the output differs slightly among browsers. The following output shows the format used by Microsoft Edge:

Console

Array(3) [11, 12, 13]

Pass data to a .NET method when calling the `invokeMethodAsync` function by passing the data as arguments.

To demonstrate passing data to .NET, pass a starting position to the `ReturnArrayAsync` method where the method is invoked in JS:

JavaScript

```
export function returnArrayAsync() {
  DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync', 14)
    .then(data => {
      console.log(data);
    });
}
```

The component's invocable `ReturnArrayAsync` method receives the starting position and constructs the array from it. The array is returned for logging to the console:

C#

```
[JSInvokable]
public static Task<int[]> ReturnArrayAsync(int startPosition) =>
    Task.FromResult(Enumerable.Range(startPosition, 3).ToArray());
```

After the app is recompiled and the browser is refreshed, the following output appears in the browser's console when the button is selected:

Console

```
Array(3) [ 14, 15, 16 ]
```

The .NET method identifier for the JS call is the .NET method name, but you can specify a different identifier using the `[JSInvokable]` attribute constructor. In the following example, `DifferentMethodName` is the assigned method identifier for the `ReturnArrayAsync` method:

C#

```
[JSInvokable("DifferentMethodName")]
```

In the call to `DotNet.invokeMethodAsync` (server-side or client-side components) or `DotNet.invokeMethod` (client-side components only), call `DifferentMethodName` to execute the `ReturnArrayAsync` .NET method:

- `DotNet.invokeMethodAsync('BlazorSample', 'DifferentMethodName');`
- `DotNet.invokeMethod('BlazorSample', 'DifferentMethodName');` (client-side components only)

ⓘ Note

The `ReturnArrayAsync` method example in this section returns the result of a [Task](#) without the use of explicit C# [async](#) and [await](#) keywords. Coding methods with [async](#) and [await](#) is typical of methods that use the [await](#) keyword to return the value of asynchronous operations.

`ReturnArrayAsync` method composed with [async](#) and [await](#) keywords:

C#

```
[JSInvokable]
public static async Task<int[]> ReturnArrayAsync() =>
    await Task.FromResult(new int[] { 11, 12, 13 });
```

For more information, see [Asynchronous programming with async and await](#) in the C# guide.

Create JavaScript object and data references to pass to .NET

Call `DotNet.createJSObjectReference(jsObject)` to construct a JS object reference so that it can be passed to .NET, where `jsObject` is the [JS Object](#) used to create the JS object reference. The following example passes a reference to the non-serializable `window` object to .NET, which receives it in the `ReceiveWindowObject` C# method as an [IJSObjectReference](#):

JavaScript

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', 'ReceiveWindowObject',  
    DotNet.createJSObjectReference(window));
```

C#

```
[JSInvokable]  
public static void ReceiveWindowObject(IJSObjectReference objRef)  
{  
    ...  
}
```

In the preceding example, the `{ASSEMBLY NAME}` placeholder is the app's namespace.

ⓘ Note

The preceding example doesn't require disposal of the `JSObjectReference`, as a reference to the `window` object isn't held in JS.

Maintaining a reference to a `JSObjectReference` requires disposing of it to avoid leaking JS memory on the client. The following example refactors the preceding code to capture a reference to the `JSObjectReference`, followed by a call to `DotNet.disposeJSObjectReference()` to dispose of the reference:

JavaScript

```
var jsObjectReference = DotNet.createJSObjectReference(window);
```

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', 'ReceiveWindowObject',  
jsObjectReference);
```

```
DotNet.disposeJSObjectReference(jsObjectReference);
```

In the preceding example, the `{ASSEMBLY NAME}` placeholder is the app's namespace.

Call `DotNet.createJSStreamReference(streamReference)` to construct a JS stream reference so that it can be passed to .NET, where `streamReference` is an [ArrayBuffer](#), [Blob](#), or any [typed array](#), such as [Uint8Array](#) or [Float32Array](#), used to create the JS stream reference.

Invoke an instance .NET method

To invoke an instance .NET method from JavaScript (JS):

- Pass the .NET instance by reference to JS by wrapping the instance in a [DotNetObjectReference](#) and calling [Create](#) on it.
- Invoke a .NET instance method from JS using `invokeMethodAsync` (*recommended*) or `invokeMethod` (client-side components only) from the passed [DotNetObjectReference](#). Pass the identifier of the instance .NET method and any arguments. The .NET instance can also be passed as an argument when invoking other .NET methods from JS.

In the following example:

- `dotNetHelper` is a [DotNetObjectReference](#).
- The `{.NET METHOD ID}` placeholder is the .NET method identifier.
- The `{ARGUMENTS}` placeholder are optional, comma-separated arguments to pass to the method, each of which must be JSON-serializable.

JavaScript

```
dotNetHelper.invokeMethodAsync('{.NET METHOD ID}', {ARGUMENTS});
```

⚠ Note

`invokeMethodAsync` and `invokeMethod` don't accept an assembly name parameter when invoking an instance method.

`invokeMethodAsync` returns a [JS Promise](#) representing the result of the operation.

`invokeMethod` (client-side components only) returns the result of the operation.

❗ Important

For server-side components, we recommend the asynchronous function (`invokeMethodAsync`) over the synchronous version (`invokeMethod`).

- Dispose of the [DotNetObjectReference](#).

The following sections of this article demonstrate various approaches for invoking an instance .NET method:

- [Pass a DotNetObjectReference to an individual JavaScript function](#)
- [Pass a DotNetObjectReference to a class with multiple JavaScript functions](#)
- [Call .NET generic class methods](#)
- [Class instance examples](#)
- [Component instance .NET method helper class](#)

Avoid trimming JavaScript-invokable .NET methods

This section applies to client-side apps with [ahead-of-time \(AOT\) compilation](#) and [runtime relinking](#) enabled.

Several of the examples in the following sections are based on a class instance approach, where the JavaScript-invokable .NET method marked with the [\[JSInvokable\]](#) attribute is a member of a class that isn't a Razor component. When such .NET methods are located in a Razor component, they're protected from [runtime relinking/trimming](#). In order to protect the .NET methods from trimming outside of Razor components, implement the methods with the [DynamicDependency attribute](#) on the class's constructor, as the following example demonstrates:

C#

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.JSInterop;

public class ExampleClass {

    [DynamicDependency(nameof(ExampleJSInvokableMethod))]
    public ExampleClass()
    {
    }
```

```

    }

    [JSInvokable]
    public string ExampleJSInvokableMethod()
    {
        ...
    }
}

```

For more information, see [Prepare .NET libraries for trimming: DynamicDependency](#).

Pass a `DotNetObjectReference` to an individual JavaScript function

The example in this section demonstrates how to pass a `DotNetObjectReference` to an individual JavaScript (JS) function.

The following `sayHello1` JS function receives a `DotNetObjectReference` and calls `invokeMethodAsync` to call the `GetHelloMessage` .NET method of a component:

HTML

```

<script>
  window.sayHello1 = (dotNetHelper) => {
    return dotNetHelper.invokeMethodAsync('GetHelloMessage');
  };
</script>

```

ⓘ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

For the following component:

- The component has a JS-invokable .NET method named `GetHelloMessage`.
- When the `Trigger .NET instance method` button is selected, the JS function `sayHello1` is called with the `DotNetObjectReference`.
- `sayHello1`:
 - Calls `GetHelloMessage` and receives the message result.

- Returns the message result to the calling `TriggerDotNetInstanceMethod` method.
- The returned message from `sayHello1` in `result` is displayed to the user.
- To avoid a memory leak and allow garbage collection, the .NET object reference created by `DotNetObjectReference` is disposed in the `Dispose` method.

`CallDotnet2.razor`:

razor

```
@page "/call-dotnet-2"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 2</PageTitle>

<h1>Call .NET Example 2</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

@code {
    private string? name;
    private string? result;
    private DotNetObjectReference<CallDotnet2>? objRef;

    protected override void OnInitialized() =>
        objRef = DotNetObjectReference.Create(this);

    public async Task TriggerDotNetInstanceMethod() =>
        result = await JS.InvokeAsync<string>("sayHello1", objRef);

    [JSInvokable]
    public string GetHelloMessage() => $"Hello, {name}!";

    public void Dispose() => objRef?.Dispose();
}
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

Use the following guidance to pass arguments to an instance method:

Add parameters to the .NET method invocation. In the following example, a name is passed to the method. Add additional parameters to the list as needed.

HTML

```
<script>
    window.sayHello2 = (dotNetHelper, name) => {
        return dotNetHelper.invokeMethodAsync('GetHelloMessage', name);
    };
</script>
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

Provide the parameter list to the .NET method.

`CallDotnet3.razor`:

razor

```
@page "/call-dotnet-3"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 3</PageTitle>

<h1>Call .NET Example 3</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

@code {
```

```

private string? name;
private string? result;
private DotNetObjectReference<CallDotnet3>? objRef;

protected override void OnInitialized() =>
    objRef = DotNetObjectReference.Create(this);

public async Task TriggerDotNetInstanceMethod() =>
    result = await JS.InvokeAsync<string>("sayHello2", objRef, name);

[JSInvokable]
public string GetHelloMessage(string passedName) => $"Hello,
{passedName}!";

public void Dispose() => objRef?.Dispose();
}

```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

Pass a `DotNetObjectReference` to a class with multiple JavaScript functions

The example in this section demonstrates how to pass a `DotNetObjectReference` to a JavaScript (JS) class with multiple functions.

Create and pass a `DotNetObjectReference` from the `OnAfterRenderAsync` lifecycle method to a JS class for multiple functions to use. Make sure that the .NET code disposes of the `DotNetObjectReference`, as the following example shows.

In the following component, the `Trigger JS function` buttons call JS functions by setting the JS `onclick` property, not Blazor's `@onclick` directive attribute.

`CallDotNetExampleOneHelper.razor`:

razor

```

@page "/call-dotnet-example-one-helper"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET Example</PageTitle>

<h1>Pass <code>DotNetObjectReference</code> to a JavaScript class</h1>

<p>
    <label>
        Message: <input @bind="name" />

```

```

        </label>
    </p>

    <p>
        <button id="sayHelloBtn">
            Trigger JS function <code>sayHello</code>
        </button>
    </p>

    <p>
        <button id="welcomeVisitorBtn">
            Trigger JS function <code>welcomeVisitor</code>
        </button>
    </p>

    @code {
        private IJSObjectReference? module;
        private string? name;
        private DotNetObjectReference<CallDotNetExampleOneHelper>? dotNetHelper;

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

                dotNetHelper = DotNetObjectReference.Create(this);
                await module.InvokeVoidAsync("GreetingHelpers.setDotNetHelper",
                    dotNetHelper);

                await module.InvokeVoidAsync("addHandlers");
            }
        }

        [JSInvokable]
        public string GetHelloMessage() => $"Hello, {name}!";

        [JSInvokable]
        public string GetWelcomeMessage() => $"Welcome, {name}!";

        async ValueTask IAsyncDisposable.DisposeAsync()
        {
            if (module is not null)
            {
                await module.DisposeAsync();
            }

            dotNetHelper?.Dispose();
        }
    }

```

In the preceding example:

- `JS` is an injected `IJSRuntime` instance. `IJSRuntime` is registered by the Blazor framework.
- The variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.
- The component must explicitly dispose of the `DotNetObjectReference` to permit garbage collection and prevent a memory leak.

`CallDotNetExampleOneHelper.razor.js`:

JavaScript

```
export class GreetingHelpers {
    static dotNetHelper;

    static setDotNetHelper(value) {
        GreetingHelpers.dotNetHelper = value;
    }

    static async sayHello() {
        const msg =
            await
GreetingHelpers.dotNetHelper.invokeMethodAsync('GetHelloMessage');
        alert(`Message from .NET: "${msg}"`);
    }

    static async welcomeVisitor() {
        const msg =
            await
GreetingHelpers.dotNetHelper.invokeMethodAsync('GetWelcomeMessage');
        alert(`Message from .NET: "${msg}"`);
    }
}

export function addHandlers() {
    const sayHelloBtn = document.getElementById("sayHelloBtn");
    sayHelloBtn.addEventListener("click", GreetingHelpers.sayHello);

    const welcomeVisitorBtn = document.getElementById("welcomeVisitorBtn");
    welcomeVisitorBtn.addEventListener("click",
GreetingHelpers.welcomeVisitor);
}
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

Call .NET generic class methods

JavaScript (JS) functions can call [.NET generic class](#) methods, where a JS function calls a .NET method of a generic class.

In the following generic type class (`GenericType<TValue>`):

- The class has a single type parameter (`TValue`) with a single generic `Value` property.
- The class has two non-generic methods marked with the [\[JSInvokable\]](#) attribute, each with a generic type parameter named `newValue`:
 - `Update` synchronously updates the value of `Value` from `newValue`.
 - `UpdateAsync` asynchronously updates the value of `Value` from `newValue` after creating an awaitable task with [Task.Yield](#) that asynchronously yields back to the current context when awaited.
- Each of the class methods write the type of `TValue` and the value of `Value` to the console. Writing to the console is only for demonstration purposes. Production apps usually avoid writing to the console in favor of app *logging*. For more information, see [ASP.NET Core Blazor logging](#) and [Logging in .NET Core and ASP.NET Core](#).

ⓘ Note

Open generic types and methods don't specify types for type placeholders. Conversely, *closed generics* supply types for all type placeholders. The examples in this section demonstrate closed generics, but invoking JS interop *instance methods* with open generics **is supported**. Use of open generics isn't supported for [static .NET method invocations](#), which were described earlier in this article.

For more information, see the following articles:

- [Generic classes and methods \(C# documentation\)](#)
- [Generic Classes \(C# Programming Guide\)](#)
- [Generics in .NET \(.NET documentation\)](#)

`GenericType.cs`:

```

using Microsoft.JSInterop;

public class GenericType<TValue>
{
    public TValue? Value { get; set; }

    [JSInvokable]
    public void Update(TValue newValue)
    {
        Value = newValue;

        Console.WriteLine($"Update: GenericType<{typeof(TValue)}>:
{Value}");
    }

    [JSInvokable]
    public async void UpdateAsync(TValue newValue)
    {
        await Task.Yield();
        Value = newValue;

        Console.WriteLine($"UpdateAsync: GenericType<{typeof(TValue)}>:
{Value}");
    }
}

```

In the following `invokeMethodsAsync` function:

- The generic type class's `Update` and `UpdateAsync` methods are called with arguments representing strings and numbers.
- Client-side components support calling .NET methods synchronously with `invokeMethod`. `syncInterop` receives a boolean value indicating if the JS interop is occurring on the client. When `syncInterop` is `true`, `invokeMethod` is safely called. If the value of `syncInterop` is `false`, only the asynchronous function `invokeMethodAsync` is called because the JS interop is executing in a server-side component.
- For demonstration purposes, the [DotNetObjectReference](#) function call (`invokeMethod` or `invokeMethodAsync`), the .NET method called (`Update` or `UpdateAsync`), and the argument are written to the console. The arguments use a random number to permit matching the JS function call to the .NET method invocation (also written to the console on the .NET side). Production code usually doesn't write to the console, either on the client or the server. Production apps usually rely upon app *logging*. For more information, see [ASP.NET Core Blazor logging](#) and [Logging in .NET Core and ASP.NET Core](#).

```

<script>
    const randomInt = () => Math.floor(Math.random() * 99999);

    window.invokeMethodsAsync = async (syncInterop, dotNetHelper1,
dotNetHelper2) => {
        var n = randomInt();
        console.log(`JS: invokeMethodAsync:Update('string ${n}')`);
        await dotNetHelper1.invokeMethodAsync('Update', `string ${n}`);

        n = randomInt();
        console.log(`JS: invokeMethodAsync:UpdateAsync('string ${n}')`);
        await dotNetHelper1.invokeMethodAsync('UpdateAsync', `string ${n}`);

        if (syncInterop) {
            n = randomInt();
            console.log(`JS: invokeMethod:Update('string ${n}')`);
            dotNetHelper1.invokeMethod('Update', `string ${n}`);
        }

        n = randomInt();
        console.log(`JS: invokeMethodAsync:Update(${n})`);
        await dotNetHelper2.invokeMethodAsync('Update', n);

        n = randomInt();
        console.log(`JS: invokeMethodAsync:UpdateAsync(${n})`);
        await dotNetHelper2.invokeMethodAsync('UpdateAsync', n);

        if (syncInterop) {
            n = randomInt();
            console.log(`JS: invokeMethod:Update(${n})`);
            dotNetHelper2.invokeMethod('Update', n);
        }
    };
</script>

```

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

In the following `GenericsExample` component:

- The JS function `invokeMethodsAsync` is called when the `Invoke Interop` button is selected.
- A pair of `DotNetObjectReference` types are created and passed to the JS function for instances of the `GenericType` as a `string` and an `int`.

`GenericsExample.razor`:


```

@page "/generics-example"
@implements IDisposable
@Inject IJSRuntime JS

<p>
    <button @onclick="InvokeInterop">Invoke Interop</button>
</p>

<ul>
    <li>genericType1: @genericType1?.Value</li>
    <li>genericType2: @genericType2?.Value</li>
</ul>

@code {
    private GenericType<string> genericType1 = new() { Value = "string 0" };
    private GenericType<int> genericType2 = new() { Value = 0 };
    private DotNetObjectReference<GenericType<string>>? objRef1;
    private DotNetObjectReference<GenericType<int>>? objRef2;

    protected override void OnInitialized()
    {
        objRef1 = DotNetObjectReference.Create(genericType1);
        objRef2 = DotNetObjectReference.Create(genericType2);
    }

    public async Task InvokeInterop()
    {
        var syncInterop = OperatingSystem.IsBrowser();

        await JS.InvokeVoidAsync(
            "invokeMethodsAsync", syncInterop, objRef1, objRef2);
    }

    public void Dispose()
    {
        objRef1?.Dispose();
        objRef2?.Dispose();
    }
}

```

In the preceding example, `JS` is an injected `IJSRuntime` instance. `IJSRuntime` is registered by the Blazor framework.

The following demonstrates typical output of the preceding example when the **Invoke Interop** button is selected in a client-side component:

```

JS: invokeMethodAsync:Update('string 37802')
.NET: Update: GenericType<System.String>: string 37802
JS: invokeMethodAsync:UpdateAsync('string 53051')

```

```
JS: invokeMethod:Update('string 26784')
.NET: Update: GenericType<System.String>: string 26784
JS: invokeMethodAsync:Update(14107)
.NET: Update: GenericType<System.Int32>: 14107
JS: invokeMethodAsync:UpdateAsync(48995)
JS: invokeMethod:Update(12872)
.NET: Update: GenericType<System.Int32>: 12872
.NET: UpdateAsync: GenericType<System.String>: string 53051
.NET: UpdateAsync: GenericType<System.Int32>: 48995
```

If the preceding example is implemented in a server-side component, the synchronous calls with `invokeMethod` are avoided. For server-side components, we recommend the asynchronous function (`invokeMethodAsync`) over the synchronous version (`invokeMethod`).

Typical output of a server-side component:

```
JS: invokeMethodAsync:Update('string 34809')
.NET: Update: GenericType<System.String>: string 34809
JS: invokeMethodAsync:UpdateAsync('string 93059')
JS: invokeMethodAsync:Update(41997)
.NET: Update: GenericType<System.Int32>: 41997
JS: invokeMethodAsync:UpdateAsync(24652)
.NET: UpdateAsync: GenericType<System.String>: string 93059
.NET: UpdateAsync: GenericType<System.Int32>: 24652
```

The preceding output examples demonstrate that asynchronous methods execute and complete in an *arbitrary order* depending on several factors, including thread scheduling and the speed of method execution. It isn't possible to reliably predict the order of completion for asynchronous method calls.

Class instance examples

The following `sayHello1` JS function:

- Calls the `GetHelloMessage` .NET method on the passed [DotNetObjectReference](#).
- Returns the message from `GetHelloMessage` to the `sayHello1` caller.

```
<script>
    window.sayHello1 = (dotNetHelper) => {
        return dotNetHelper.invokeMethodAsync('GetHelloMessage');
    };
</script>
```

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

The following `HelloHelper` class has a JS-invokable .NET method named `GetHelloMessage`. When `HelloHelper` is created, the name in the `Name` property is used to return a message from `GetHelloMessage`.

`HelloHelper.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class HelloHelper(string? name)
{
    public string? Name { get; set; } = name ?? "No Name";

    [JSInvokable]
    public string GetHelloMessage() => $"Hello, {Name}!";
}
```

The `CallHelloHelperGetHelloMessage` method in the following `JsInteropClasses3` class invokes the JS function `sayHello1` with a new instance of `HelloHelper`.

`JsInteropClasses3.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;
```

```

public class JsInteropClasses3(IJSRuntime js)
{
    private readonly IJSRuntime js = js;

    public async ValueTask<string> CallHelloHelperGetHelloMessage(string?
name)
    {
        using var objRef = DotNetObjectReference.Create(new
HelloHelper(name));
        return await js.InvokeAsync<string>("sayHello1", objRef);
    }
}

```

To avoid a memory leak and allow garbage collection, the .NET object reference created by `DotNetObjectReference` is disposed when the object reference goes out of scope with `using var` syntax.

When the **Trigger .NET instance method** button is selected in the following component, `JsInteropClasses3.CallHelloHelperGetHelloMessage` is called with the value of `name`.

CallDotnet4.razor:

razor

```

@page "/call-dotnet-4"
@inject IJSRuntime JS

<PageTitle>Call .NET 4</PageTitle>

<h1>Call .NET Example 4</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

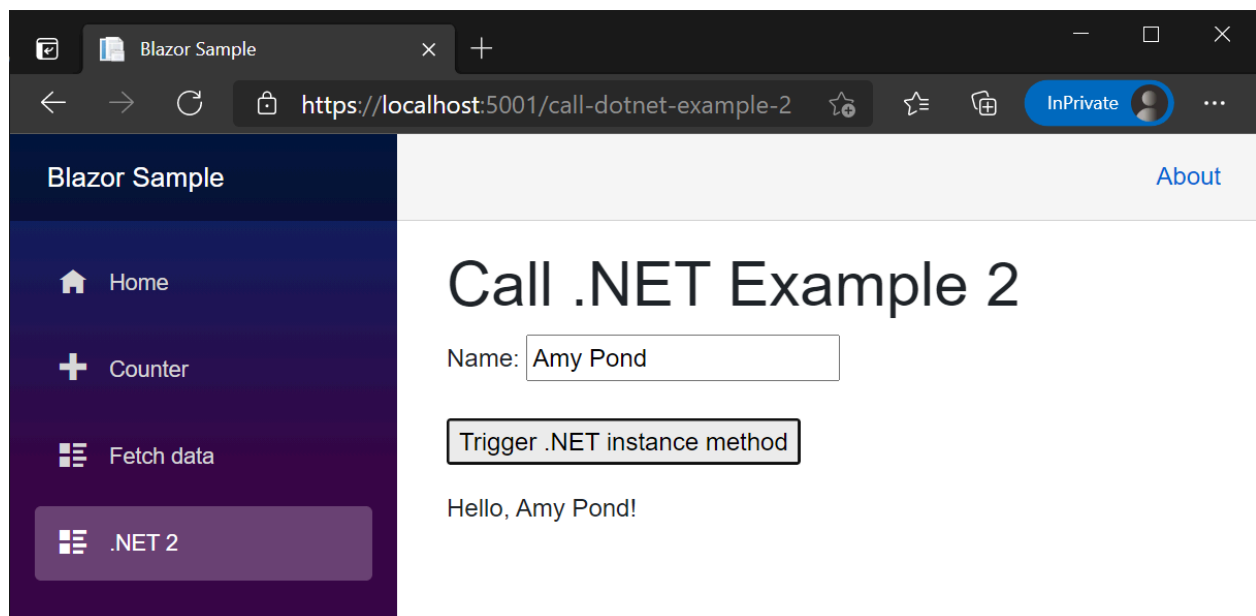
@code {
    private string? name;
    private string? result;
    private JsInteropClasses3? jsInteropClasses;
}

```

```
protected override void OnInitialized() =>
    jsInteropClasses = new JsInteropClasses3(JS);

private async Task TriggerDotNetInstanceMethod()
{
    if (jsInteropClasses is not null)
    {
        result = await
jsInteropClasses.CallHelloHelperGetHelloMessage(name);
    }
}
}
```

The following image shows the rendered component with the name `Amy Pond` in the `Name` field. After the button is selected, `Hello, Amy Pond!` is displayed in the UI:



The preceding pattern shown in the `JsInteropClasses3` class can also be implemented entirely in a component.

`CallDotnet5.razor`:

razor

```
@page "/call-dotnet-5"
@inject IJSRuntime JS

<PageTitle>Call .NET 5</PageTitle>

<h1>Call .NET Example 5</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>
```

```

</p>

<p>
  <button @onclick="TriggerDotNetInstanceMethod">
    Trigger .NET instance method
  </button>
</p>

<p>
  @result
</p>

@code {
  private string? name;
  private string? result;

  public async Task TriggerDotNetInstanceMethod()
  {
    using var objRef = DotNetObjectReference.Create(new
HelloHelper(name));
    result = await JS.InvokeAsync<string>("sayHello1", objRef);
  }
}

```

To avoid a memory leak and allow garbage collection, the .NET object reference created by [DotNetObjectReference](#) is disposed when the object reference goes out of scope with [using var syntax](#).

The output displayed by the component is `Hello, Amy Pond!` when the name `Amy Pond` is provided in the `name` field.

In the preceding component, the .NET object reference is disposed. If a class or component doesn't dispose the [DotNetObjectReference](#), dispose it from the client by calling `dispose` on the passed [DotNetObjectReference](#):

JavaScript

```

window.{JS FUNCTION NAME} = (dotNetHelper) => {
  dotNetHelper.invokeMethodAsync('{.NET METHOD ID}');
  dotNetHelper.dispose();
}

```

In the preceding example:

- The `{JS FUNCTION NAME}` placeholder is the JS function's name.
- The variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.
- The `{.NET METHOD ID}` placeholder is the .NET method identifier.

Component instance .NET method helper class

A helper class can invoke a .NET instance method as an [Action](#). Helper classes are useful in scenarios where using static .NET methods aren't applicable:

- When several components of the same type are rendered on the same page.
- In server-side apps with multiple users concurrently using the same component.

In the following example:

- The component contains several `ListItem1` components.
- Each `ListItem1` component is composed of a message and a button.
- When a `ListItem1` component button is selected, that `ListItem1`'s `UpdateMessage` method changes the list item text and hides the button.

The following `MessageUpdateInvokeHelper` class maintains a JS-invokable .NET method, `UpdateMessageCaller`, to invoke the [Action](#) specified when the class is instantiated.

`MessageUpdateInvokeHelper.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class MessageUpdateInvokeHelper(Action action)
{
    private readonly Action action = action;

    [JSInvokable]
    public void UpdateMessageCaller() => action.Invoke();
}
```

The following `updateMessageCaller` JS function invokes the `UpdateMessageCaller` .NET method.

HTML

```
<script>
    window.updateMessageCaller = (dotNetHelper) => {
        dotNetHelper.invokeMethodAsync('UpdateMessageCaller');
        dotNetHelper.dispose();
    }
</script>
```

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

The following `ListItem1` component is a shared component that can be used any number of times in a parent component and creates list items (`...`) for an HTML list (`...` or `...`). Each `ListItem1` component instance establishes an instance of `MessageUpdateInvokeHelper` with an `Action` set to its `UpdateMessage` method.

When a `ListItem1` component's `InteropCall` button is selected, `updateMessageCaller` is invoked with a created `DotNetObjectReference` for the `MessageUpdateInvokeHelper` instance. This permits the framework to call `UpdateMessageCaller` on that `ListItem1`'s `MessageUpdateInvokeHelper` instance. The passed `DotNetObjectReference` is disposed in JS (`dotNetHelper.dispose()`).

`ListItem1.razor`:

razor

```
@inject IJSRuntime JS

<li>
    @message
    <button @onclick="InteropCall"
style="display:@display">InteropCall</button>
</li>

@code {
    private string message = "Select one of these list item buttons.";
    private string display = "inline-block";
    private MessageUpdateInvokeHelper? messageUpdateInvokeHelper;

    protected override void OnInitialized()
    {
        messageUpdateInvokeHelper = new
MessageUpdateInvokeHelper(UpdateMessage);
    }

    protected async Task InteropCall()
    {
        if (messageUpdateInvokeHelper is not null)
        {

```



```

        await JS.InvokeVoidAsync("updateMessageCaller",
            DotNetObjectReference.Create(messageUpdateInvokeHelper));
    }
}

private void UpdateMessage()
{
    message = "UpdateMessage Called!";
    display = "none";
    StateHasChanged();
}
}

```

`StateHasChanged` is called to update the UI when `message` is set in `UpdateMessage`. If `StateHasChanged` isn't called, Blazor has no way of knowing that the UI should be updated when the [Action](#) is invoked.

The following parent component includes four list items, each an instance of the `ListItems1` component.

`CallDotnet6.razor`:

```

razor

@page "/call-dotnet-6"

<PageTitle>Call .NET 6</PageTitle>

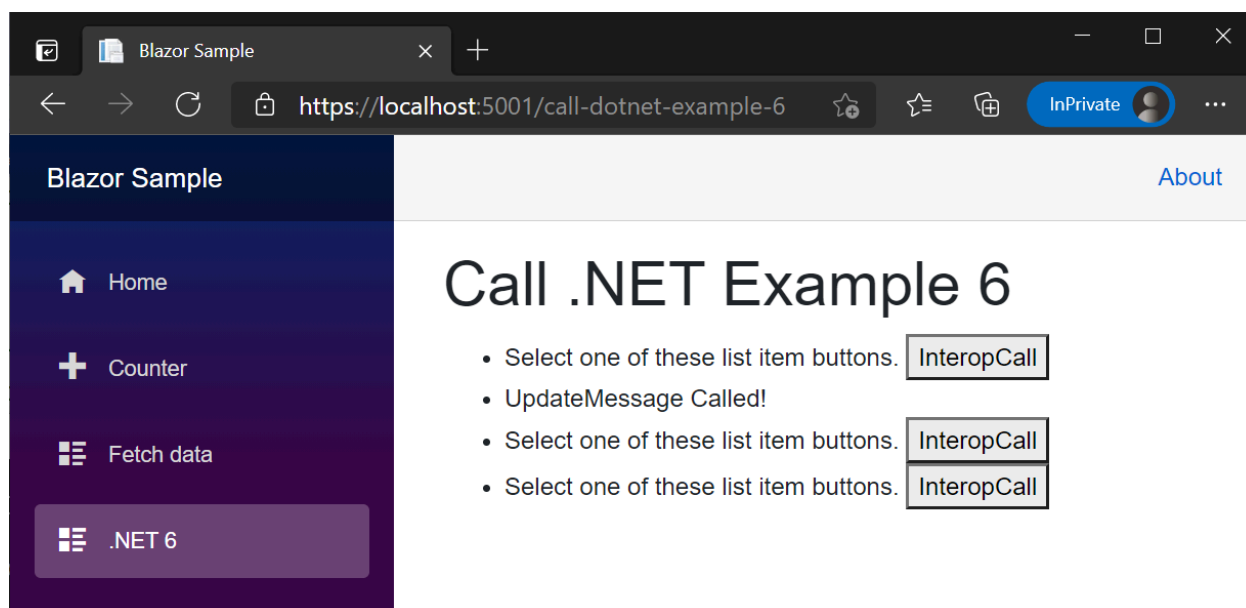
<h1>Call .NET Example 6</h1>

<ul>
    <ListItems1 />
    <ListItems1 />
    <ListItems1 />
    <ListItems1 />
</ul>

```

The following image shows the rendered parent component after the second `InteropCall` button is selected:

- The second `ListItems1` component has displayed the `UpdateMessage Called!` message.
- The `InteropCall` button for the second `ListItems1` component isn't visible because the button's CSS `display` property is set to `none`.



Component instance .NET method called from `DotNetObjectReference` assigned to an element property

The assignment of a `DotNetObjectReference` to a property of an HTML element permits calling .NET methods on a component instance:

- An `element reference` is captured (`ElementReference`).
- In the component's `OnAfterRender{Async}` method, a JavaScript (JS) function is invoked with the element reference and the component instance as a `DotNetObjectReference`. The JS function attaches the `DotNetObjectReference` to the element in a property.
- When an element event is invoked in JS (for example, `onclick`), the element's attached `DotNetObjectReference` is used to call a .NET method.

Similar to the approach described in the [Component instance .NET method helper class](#) section, this approach is useful in scenarios where using static .NET methods aren't applicable:

- When several components of the same type are rendered on the same page.
- In server-side apps with multiple users concurrently using the same component.
- The .NET method is invoked from a JS event (for example, `onclick`), not from a Blazor event (for example, `@onclick`).

In the following example:

- The component contains several `ListItem2` components, which is a shared component.

- Each `ListItem2` component is composed of a list item message `` and a second `` with a `display` CSS property set to `inline-block` for display.
- When a `ListItem2` component list item is selected, that `ListItem2`'s `UpdateMessage` method changes the list item text in the first `` and hides the second `` by setting its `display` property to `none`.

The following `assignDotNetHelper` JS function assigns the [DotNetObjectReference](#) to an element in a property named `dotNetHelper`. The following `interopCall` JS function uses the [DotNetObjectReference](#) for the passed element to invoke a .NET method named `UpdateMessage`.

`ListItem2.razor.js`:

JavaScript

```
export function assignDotNetHelper(element, dotNetHelper) {
    element.dotNetHelper = dotNetHelper;
}

export async function interopCall(element) {
    await element.dotNetHelper.invokeMethodAsync('UpdateMessage');
}
```

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

The following `ListItem2` component is a shared component that can be used any number of times in a parent component and creates list items (`...`) for an HTML list (`...` or `...`).

Each `ListItem2` component instance invokes the `assignDotNetHelper` JS function in [OnAfterRenderAsync](#) with an element reference (the first `` element of the list item) and the component instance as a [DotNetObjectReference](#).

When a `ListItem2` component's message `` is selected, `interopCall` is invoked passing the `` element as a parameter (`this`), which invokes the `UpdateMessage` .NET method. In `UpdateMessage`, [StateHasChanged](#) is called to update the UI when

`message` is set and the `display` property of the second `` is updated. If `StateHasChanged` isn't called, Blazor has no way of knowing that the UI should be updated when the method is invoked.

The [DotNetObjectReference](#) is disposed when the component is disposed.

`ListItem2.razor`:

razor

```
@inject IJSRuntime JS
@implements IAsyncDisposable

<li>
    <span style="font-weight:bold;color:@color" @ref="elementRef"
        @onclick="CallJSToInvokeDotnet">
        @message
    </span>
    <span style="display:@display">
        Not Updated Yet!
    </span>
</li>

@code {
    private IJSObjectReference? module;
    private DotNetObjectReference<ListItem2>? objRef;
    private ElementReference elementRef;
    private string display = "inline-block";
    private string message = "Select one of these list items.";
    private string color = "initial";

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/ListItem2.razor.js");

            objRef = DotNetObjectReference.Create(this);
            await module.InvokeVoidAsync("assignDotNetHelper", elementRef,
objRef);
        }
    }

    public async void CallJSToInvokeDotnet()
    {
        if (module is not null)
        {
            await module.InvokeVoidAsync("interopCall", elementRef);
        }
    }

    [JSInvokable]
```

```

public void UpdateMessage()
{
    message = "UpdateMessage Called!";
    display = "none";
    color = "MediumSeaGreen";
    StateHasChanged();
}

async ValueTask IAsyncDisposable.DisposeAsync()
{
    if (module is not null)
    {
        await module.DisposeAsync();
    }

    objRef?.Dispose();
}
}

```

The following parent component includes four list items, each an instance of the `ListItem2` component.

`CallDotnet7.razor`:

razor

```

@page "/call-dotnet-7"

<PageTitle>Call .NET 7</PageTitle>

<h1>Call .NET Example 7</h1>

<ul>
    <ListItem2 />
    <ListItem2 />
    <ListItem2 />
    <ListItem2 />
</ul>

```

Synchronous JS interop in client-side components

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible

across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from JavaScript to .NET in a client-side component, use `DotNet.invokeMethod` instead of `DotNet.invokeMethodAsync`.

Synchronous calls work if:

- The component is only rendered for execution on WebAssembly.
- The called function returns a value synchronously. The function isn't an `async` method and doesn't return a .NET [Task](#) or JavaScript [Promise](#) [↗].

JavaScript location

Load JavaScript (JS) code using any of approaches described by the [article on JavaScript location](#):

- [Load a script in <head> markup](#) (*Not generally recommended*)
- [Load a script in <body> markup](#)
- [Load a script from an external JavaScript file \(.js\) collocated with a component](#)
- [Load a script from an external JavaScript file \(.js\)](#)
- [Inject a script before or after Blazor starts](#)

Using JS modules to load JS is described in this article in the [JavaScript isolation in JavaScript modules](#) section.

Warning

Only place a `<script>` tag in a component file (`.razor`) if the component is guaranteed to adopt **static server-side rendering (static SSR)** because the `<script>` tag can't be updated dynamically.

JavaScript isolation in JavaScript modules

Blazor enables JavaScript (JS) isolation in standard [JavaScript modules](#) [↗] ([ECMAScript specification](#) [↗]). JavaScript module loading works the same way in Blazor as it does for

other types of web apps, and you're free to customize how modules are defined in your app. For a guide on how to use JavaScript modules, see [MDN Web Docs: JavaScript modules](#).

JS isolation provides the following benefits:

- Imported JS no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JS.

For more information, see [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#).

[Dynamic import with the import\(\) operator](#) is supported with ASP.NET Core and Blazor:

```
JavaScript

if ({CONDITION}) import("/additionalModule.js");
```

In the preceding example, the `{CONDITION}` placeholder represents a conditional check to determine if the module should be loaded.

For browser compatibility, see [Can I use: JavaScript modules: dynamic import](#).

Avoid circular object references

Objects that contain circular references can't be serialized on the client for either:

- .NET method calls.
- JavaScript method calls from C# when the return type has circular references.

Byte array support

Blazor supports optimized byte array JavaScript (JS) interop that avoids encoding/decoding byte arrays into Base64. The following example uses JS interop to pass a byte array to .NET.

Provide a `sendByteArray` JS function. The function is called statically, which includes the assembly name parameter in the `invokeMethodAsync` call, by a button in the component and doesn't return a value:

```
CallDotnet8.razor.js:
```

```

export function sendByteArray() {
    const data = new Uint8Array([0x45, 0x76, 0x65, 0x72, 0x79, 0x74, 0x68,
0x69,
    0x6e, 0x67, 0x27, 0x73, 0x20, 0x73, 0x68, 0x69, 0x6e, 0x79, 0x2c,
    0x20, 0x43, 0x61, 0x70, 0x74, 0x61, 0x69, 0x6e, 0x2e, 0x20, 0x4e,
    0x6f, 0x74, 0x20, 0x74, 0x6f, 0x20, 0x66, 0x72, 0x65, 0x74, 0x2e]);
    DotNet.invokeMethodAsync('BlazorSample', 'ReceiveByteArray', data)
        .then(str => {
            alert(str);
        });
}

export function addHandlers() {
    const btn = document.getElementById("btn");
    btn.addEventListener("click", sendByteArray);
}

```

❗ Note

For general guidance on JS location and our recommendations for production apps, see [JavaScript location in ASP.NET Core Blazor apps](#).

CallDotnet8.razor:

razor

```

@page "/call-dotnet-8"
@using System.Text
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 8</PageTitle>

<h1>Call .NET Example 8</h1>

<p>
    <button id="btn">Send Bytes</button>
</p>

<p>
    Quote ©2005 <a href="https://www.uphe.com">Universal Pictures</a>:
    <a href="https://www.uphe.com/movies/serenity-2005">Serenity</a><br>
    <a href="https://www.imdb.com/name/nm0821612/">Jewel Staite on IMDB</a>
</p>

@code {
    private IJSObjectReference? module;

```



```

protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        module = await JS.InvokeAsync<IJSObjectReference>("import",
            "./Components/Pages/CallDotnet8.razor.js");

        await module.InvokeVoidAsync("addHandlers");
    }
}

[JSInvokable]
public static Task<string> ReceiveByteArray(byte[] receivedBytes) =>
    Task.FromResult(Encoding.UTF8.GetString(receivedBytes, 0,
        receivedBytes.Length));

async ValueTask IAsyncDisposable.DisposeAsync()
{
    if (module is not null)
    {
        await module.DisposeAsync();
    }
}
}

```

For information on using a byte array when calling JavaScript from .NET, see [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#).

Stream from JavaScript to .NET

Blazor supports streaming data directly from JavaScript to .NET. Streams are requested using the `Microsoft.JSInterop.IJSStreamReference` interface.

`Microsoft.JSInterop.IJSStreamReference.OpenReadStreamAsync` returns a [Stream](#) and uses the following parameters:

- `maxAllowedSize`: Maximum number of bytes permitted for the read operation from JavaScript, which defaults to 512,000 bytes if not specified.
- `cancellation_token`: A [Cancellation Token](#) for cancelling the read.

In JavaScript:

```

JavaScript

function streamToDotNet() {
    return new Uint8Array(10000000);
}

```

In C# code:

```
C#

var dataReference =
    await JS.InvokeAsync<IJSStreamReference>("streamToDotNet");
using var dataReferenceStream =
    await dataReference.OpenReadStreamAsync(maxAllowedSize: 10_000_000);

var outputPath = Path.Combine(Path.GetTempPath(), "file.txt");
using var outputFileStream = File.OpenWrite(outputPath);
await dataReferenceStream.CopyToAsync(outputFileStream);
```

In the preceding example:

- `JS` is an injected `IJSRuntime` instance. `IJSRuntime` is registered by the Blazor framework.
- The `dataReferenceStream` is written to disk (`file.txt`) at the current user's temporary folder path (`GetTempPath`).

[Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#) covers the reverse operation, streaming from .NET to JavaScript using a `DotNetStreamReference`.

[ASP.NET Core Blazor file uploads](#) covers how to upload a file in Blazor. For a forms example that streams `<textarea>` data in a server-side component, see [Troubleshoot ASP.NET Core Blazor forms](#).

JavaScript `[JSImport]` / `[JSExport]` interop

This section applies to client-side components.

As an alternative to interacting with JavaScript (JS) in client-side components using Blazor's JS interop mechanism based on the `IJSRuntime` interface, a JS `[JSImport]` / `[JSExport]` interop API is available to apps targeting .NET 7 or later.

For more information, see [JavaScript JSImport/JSExport interop with ASP.NET Core Blazor](#).

Disposal of JavaScript interop object references

Examples throughout the JavaScript (JS) interop articles demonstrate typical object disposal patterns:

- When calling .NET from JS, as described in this article, dispose of a created [DotNetObjectReference](#) either from .NET or from JS to avoid leaking .NET memory.
- When calling JS from .NET, as described in [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#), dispose any created [IJSObjectReference/IJSInProcessObjectReference](#)/[JSObjectReference](#) either from .NET or from JS to avoid leaking JS memory.

JS interop object references are implemented as a map keyed by an identifier on the side of the JS interop call that creates the reference. When object disposal is initiated from either the .NET or JS side, Blazor removes the entry from the map, and the object can be garbage collected as long as no other strong reference to the object is present.

At a minimum, always dispose objects created on the .NET side to avoid leaking .NET managed memory.

DOM cleanup tasks during component disposal

For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

JavaScript interop calls without a circuit

For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

Additional resources

- [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#)
- [InteropComponent.razor example \(dotnet/AspNetCore GitHub repository main branch\)](#) [↗]: The `main` branch represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release (for example, `release/{VERSION}`, where the `{VERSION}` placeholder is the release version), use the **Switch branches or tags** dropdown list to select the branch. For a branch that no longer exists, use the **Tags** tab to find the API (for example, `v7.0.0`).
- [Interaction with the DOM](#)
- [Blazor samples GitHub repository \(dotnet/blazor-samples\)](#) [↗] (how to download)
- [Handle errors in ASP.NET Core Blazor apps](#) (*JavaScript interop* section)
- [Threat mitigation: .NET methods invoked from the browser](#)

JavaScript `[JSImport]` / `[JSExport]` interop with ASP.NET Core Blazor

Article • 10/04/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to interact with JavaScript (JS) in client-side components using JavaScript (JS) `[JSImport]` / `[JSExport]` interop API. For additional information and examples, see [JavaScript `\[JSImport\]` / `\[JSExport\]` interop in .NET WebAssembly](#).

For additional guidance, see the [Configuring and hosting .NET WebAssembly applications](#) guidance in the .NET Runtime (`dotnet/runtime`) GitHub repository.

Blazor provides its own JS interop mechanism based on the [IJSRuntime](#) interface. Blazor's JS interop is uniformly supported across Blazor render modes and for Blazor Hybrid apps. [IJSRuntime](#) also enables library authors to build JS interop libraries for sharing across the Blazor ecosystem and remains the recommended approach for JS interop in Blazor. See the following articles:

- [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#)
- [Call .NET methods from JavaScript functions in ASP.NET Core Blazor](#)

This article describes an alternative JS interop approach specific to client-side components executed on WebAssembly. These approaches are appropriate when you only expect to run on client-side WebAssembly. Library authors can use these approaches to optimize JS interop by checking during code execution if the app is running on WebAssembly in a browser ([OperatingSystem.IsBrowser](#)). The approaches described in this article should be used to replace the obsolete unmarshalled JS interop API when migrating to .NET 7 or later.

Note

This article focuses on JS interop in client-side components. For guidance on calling .NET in JavaScript apps, see [JavaScript `\[JSImport\]` / `\[JSExport\]` interop with a](#)

Obsolete JavaScript interop API

Unmarshalled JS interop using [JSUnmarshalledRuntime](#) API is obsolete in ASP.NET Core in .NET 7 or later. Follow the guidance in this article to replace the obsolete API.

Prerequisites

[Visual Studio](#) with the **ASP.NET and web development** workload.

No further tooling is required if you plan on implementing `[JSImport]` / `[JSExport]` interop in a Blazor WebAssembly app generated from the Blazor WebAssembly project template.

If you plan to use the WebAssembly Browser or WebAssembly Console app project templates, install the [Microsoft.NET.Runtime.WebAssembly.Templates](#) [↗](#) NuGet package with the following command:

.NET CLI

```
dotnet new install Microsoft.NET.Runtime.WebAssembly.Templates
```

For more information, see [JavaScript `\[JSImport\]`/`\[JSExport\]` interop with a WebAssembly Browser App project.](#)

Namespace

The JS interop API ([JSHost.ImportAsync](#)) described in this article is controlled by attributes in the [System.Runtime.InteropServices.JavaScript](#) namespace.

Enable unsafe blocks

Enable the [AllowUnsafeBlocks](#) property in app's project file, which permits the code generator in the Roslyn compiler to use pointers for JS interop:

XML

```
<PropertyGroup>  
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

⚠ Warning

The JS interop API requires enabling [AllowUnsafeBlocks](#). Be careful when implementing your own unsafe code in .NET apps, which can introduce security and stability risks. For more information, see [Unsafe code, pointer types, and function pointers](#).

Razor class library (RCL) collocated JS is unsupported

Generally, the JS location support for [IJSRuntime](#)-based JS interop ([JavaScript location in ASP.NET Core Blazor apps](#)) is also present for the `[JSImport]`/`[JSEExport]` interop described by this article. The only unsupported JS location feature is for *collocated JS in a Razor class library (RCL)*.

Instead of using collocated JS in an RCL, place the JS file in the RCL's `wwwroot` folder and reference it using the usual path for RCL static assets:

```
_content/{PACKAGE ID}/{PATH}/{FILE NAME}.js
```

- The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library).
- The `{PATH}` placeholder is the path to the file.
- The `{FILE NAME}` placeholder is the file name.

Although collocated JS in an RCL isn't supported by `[JSImport]`/`[JSEExport]` interop, you can keep your JS files organized by taking either or both of the following approaches:

- Name the JS file the same as the component where the JS is used. For a component in the RCL named `CallJavaScriptFromLib` (`CallJavaScriptFromLib.razor`), name the file `CallJavaScriptFromLib.js` in the `wwwroot` folder.
- Place component-specific JS files in a `Components` folder inside the RCL's `wwwroot` folder and use "Components" in the path to the file: `_content/{PACKAGE ID}/Components/CallJavaScriptFromLib.js`.

Call JavaScript from .NET

This section explains how to call JS functions from .NET.

In the following `CallJavaScript1` component:

- The `CallJavaScript1` module is imported asynchronously from the [collocated JS file](#) with `JSHost.ImportAsync`.
- The imported `getMessage` JS function is called by `GetWelcomeMessage`.
- The returned welcome message string is displayed in the UI via the `message` field.

`CallJavaScript1.razor`:

```
razor

@page "/call-javascript-1"
@rendermode InteractiveWebAssembly
@using System.Runtime.InteropServices.JavaScript

<h1>
    JS [JSImport]/[JSEExport] Interop
    (Call JS Example 1)
</h1>

@(message is not null ? message : string.Empty)

@code {
    private string? message;

    protected override async Task OnInitializedAsync()
    {
        await JSHost.ImportAsync("CallJavaScript1",
            "../Components/Pages/CallJavaScript1.razor.js");

        message = GetWelcomeMessage();
    }
}
```

ⓘ Note

Include a conditional check in code with `OperatingSystem.IsBrowser` to ensure that the JS interop is only called by a component rendered on the client. This is important for libraries/NuGet packages that target server-side components, which can't execute the code provided by this JS interop API.

To import a JS function to call it from C#, use the `[JSImport]` attribute on a C# method signature that matches the JS function's signature. The first parameter to the `[JSImport]`

attribute is the name of the JS function to import, and the second parameter is the name of the [JS module](#).

In the following example, `getMessage` is a JS function that returns a `string` for a module named `CallJavaScript1`. The C# method signature matches: No parameters are passed to the JS function, and the JS function returns a `string`. The JS function is called by `GetWelcomeMessage` in C# code.

`CallJavaScript1.razor.cs`:

C#

```
using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;

namespace BlazorSample.Components.Pages;

[SupportedOSPlatform("browser")]
public partial class CallJavaScript1
{
    [JSImport("getMessage", "CallJavaScript1")]
    internal static partial string GetWelcomeMessage();
}
```

The app's namespace for the preceding `CallJavaScript1` partial class is `BlazorSample`. The component's namespace is `BlazorSample.Components.Pages`. If using the preceding component in a local test app, update the namespace to match the app. For example, the namespace is `ContosoApp.Components.Pages` if the app's namespace is `ContosoApp`. For more information, see [ASP.NET Core Razor components](#).

In the imported method signature, you can use .NET types for parameters and return values, which are marshalled automatically by the runtime. Use [JSImportAsAttribute<T>](#) to control how the imported method parameters are marshalled. For example, you might choose to marshal a `long` as [System.Runtime.InteropServices.JavaScript.JSType.Number](#) or [System.Runtime.InteropServices.JavaScript.JSType.BigInt](#). You can pass [Action/Func<TResult>](#) callbacks as parameters, which are marshalled as callable JS functions. You can pass both JS and managed object references, and they are marshalled as proxy objects, keeping the object alive across the boundary until the proxy is garbage collected. You can also import and export asynchronous methods with a [Task](#) result, which are marshalled as [JS promises](#) [↗](#). Most of the marshalled types work in both directions, as parameters and as return values, on both imported and exported methods, which are covered in the [Call .NET from JavaScript](#) section later in this article.

For additional type mapping information and examples, see [JavaScript`\[JSImport\]`/\[JSExport\]` interop in .NET WebAssembly](#).

The module name in the `[JSImport]` attribute and the call to load the module in the component with `JSHost.ImportAsync` must match and be unique in the app. When authoring a library for deployment in a NuGet package, we recommend using the NuGet package namespace as a prefix in module names. In the following example, the module name reflects the `Contoso.InteropServices.JavaScript` package and a folder of user message interop classes (`UserMessages`):

C#

```
[JSImport("getMessage",  
    "Contoso.InteropServices.JavaScript.UserMessages.CallJavaScript1")]
```

Functions accessible on the global namespace can be imported by using the [globalThis](#) prefix in the function name and by using the `[JSImport]` attribute without providing a module name. In the following example, [console.log](#) is prefixed with `globalThis`. The imported function is called by the C# `Log` method, which accepts a C# string message (`message`) and marshalls the C# string to a JS [String](#) for `console.log`:

C#

```
[JSImport("globalThis.console.log")]  
internal static partial void Log([JSMarshalAs<JSType.String>] string  
message);
```

Export scripts from a standard [JavaScript module](#) either [collocated with a component](#) or placed with other JavaScript static assets in a JS file (for example, `wwwroot/js/{FILE NAME}.js`, where JS static assets are maintained in a folder named `js` in the app's `wwwroot` folder and the `{FILE NAME}` placeholder is the file name).

In the following example, a JS function named `getMessage` is exported from a collocated JS file that returns a welcome message, "Hello from Blazor!" in Portuguese:

`CallJavaScript1.razor.js`:

JavaScript

```
export function getMessage() {  
    return 'Olá do Blazor!';  
}
```

Call .NET from JavaScript

This section explains how to call .NET methods from JS.

The following `CallDotNet1` component calls JS that directly interacts with the DOM to render the welcome message string:

- The `CallDotNet` JS module is imported asynchronously from the collocated JS file for this component.
- The imported `setMessage` JS function is called by `SetWelcomeMessage`.
- The returned welcome message is displayed by `setMessage` in the UI via the `message` field.

Important

In this section's example, JS interop is used to mutate a DOM element *purely for demonstration purposes* after the component is rendered in [OnAfterRender](#).

Typically, you should only mutate the DOM with JS when the object doesn't interact with Blazor. The approach shown in this section is similar to cases where a third-party JS library is used in a Razor component, where the component interacts with the JS library via JS interop, the third-party JS library interacts with part of the DOM, and Blazor isn't involved directly with the DOM updates to that part of the DOM.

For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

`CallDotNet1.razor`:

razor

```
@page "/call-dotnet-1"
@rendermode InteractiveWebAssembly
@using System.Runtime.InteropServices.JavaScript

<h1>
    JS [JSImport]/[JSExport] Interop
    (Call .NET Example 1)
</h1>

<p>
    <span id="result">.NET method not executed yet</span>
</p>

@code {
    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
```

```

        if (firstRender)
        {
            await JSHost.ImportAsync("CallDotNet1",
                "../Components/Pages/CallDotNet1.razor.js");

            SetWelcomeMessage();
        }
    }
}

```

To export a .NET method so that it can be called from JS, use the [\[JSExport\] attribute](#).

In the following example:

- `SetWelcomeMessage` calls a JS function named `setMessage`. The JS function calls into .NET to receive the welcome message from `GetMessageFromDotnet` and displays the message in the UI.
- `GetMessageFromDotnet` is a .NET method with the `[JSExport]` attribute that returns a welcome message, "Hello from Blazor!" in Portuguese.

`CallDotNet1.razor.cs`:

C#

```

using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;

namespace BlazorSample.Components.Pages;

[SupportedOSPlatform("browser")]
public partial class CallDotNet1
{
    [JSImport("setMessage", "CallDotNet1")]
    internal static partial void SetWelcomeMessage();

    [JSExport]
    internal static string GetMessageFromDotnet() => "Olá do Blazor!";
}

```

The app's namespace for the preceding `CallDotNet1` partial class is `BlazorSample`. The component's namespace is `BlazorSample.Components.Pages`. If using the preceding component in a local test app, update the app's namespace to match the app. For example, the component namespace is `ContosoApp.Components.Pages` if the app's namespace is `ContosoApp`. For more information, see [ASP.NET Core Razor components](#).

In the following example, a JS function named `setMessage` is imported from a collocated JS file.

The `setMessage` method:

- Calls `globalThis.getDotnetRuntime(0)` to expose the WebAssembly .NET runtime instance for calling exported .NET methods.
- Obtains the app assembly's JS exports. The name of the app's assembly in the following example is `BlazorSample`.
- Calls the `BlazorSample.Components.Pages.CallDotNet1.GetMessageFromDotnet` method from the exports (`exports`). The returned value, which is the welcome message, is assigned to the `CallDotNet1` component's `` text. The app's namespace is `BlazorSample`, and the `CallDotNet1` component's namespace is `BlazorSample.Components.Pages`.

`CallDotNet1.razor.js`:

JavaScript

```
export async function setMessage() {
  const { getAssemblyExports } = await globalThis.getDotnetRuntime(0);
  var exports = await getAssemblyExports("BlazorSample.dll");

  document.getElementById("result").innerText =

exports.BlazorSample.Components.Pages.CallDotNet1.GetMessageFromDotnet();
}
```

⚠ Note

Calling `getAssemblyExports` to obtain the exports can occur in a [JavaScript initializer](#) for availability across the app.

Multiple module import calls

After a JS module is loaded, the module's JS functions are available to the app's components and classes as long as the app is running in the browser window or tab without the user manually reloading the app. `JSHost.ImportAsync` can be called multiple times on the same module without a significant performance penalty when:

- The user visits a component that calls `JSHost.ImportAsync` to import a module, navigates away from the component, and then returns to the component where `JSHost.ImportAsync` is called again for the same module import.
- The same module is used by different components and loaded by `JSHost.ImportAsync` in each of the components.

Use of a single JavaScript module across components

Before following the guidance in this section, read the [Call JavaScript from .NET](#) and [Call .NET from JavaScript](#) sections of this article, which provide general guidance on `[JSImport]`/`[JSExport]` interop.

The example in this section shows how to use JS interop from a shared JS module in a client-side app. The guidance in this section isn't applicable to Razor class libraries (RCLs).

The following components, classes, C# methods, and JS functions are used:

- `Interop` class (`Interop.cs`): Sets up import and export JS interop with the `[JSImport]` and `[JSExport]` attributes for a module named `Interop`.
 - `GetWelcomeMessage`: .NET method that calls the imported `getMessage` JS function.
 - `SetWelcomeMessage`: .NET method that calls the imported `setMessage` JS function.
 - `GetMessageFromDotnet`: An exported C# method that returns a welcome message string when called from JS.
- `wwwroot/js/interop.js` file: Contains the JS functions.
 - `getMessage`: Returns a welcome message when called by C# code in a component.
 - `setMessage`: Calls the `GetMessageFromDotnet` C# method and assigns the returned welcome message to a DOM `` element.
- `Program.cs` calls `JSHost.ImportAsync` to load the module from `wwwroot/js/interop.js`.
- `CallJavaScript2` component (`CallJavaScript2.razor`): Calls `GetWelcomeMessage` and displays the returned welcome message in the component's UI.
- `CallDotNet2` component (`CallDotNet2.razor`): Calls `SetWelcomeMessage`.

`Interop.cs`:

```
C#

using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;

namespace BlazorSample.JavaScriptInterop;

[SupportedOSPlatform("browser")]
public partial class Interop
{
    [JSImport("getMessage", "Interop")]
```

```

        internal static partial string GetWelcomeMessage();

        [JSImport("setMessage", "Interop")]
        internal static partial void SetWelcomeMessage();

        [JSEExport]
        internal static string GetMessageFromDotnet() => "Olá do Blazor!";
    }

```

In the preceding example, the app's namespace is `BlazorSample`, and the full namespace for C# interop classes is `BlazorSample.JavaScriptInterop`.

`wwwroot/js/interop.js`:

JavaScript

```

export function getMessage() {
    return 'Olá do Blazor!';
}

export async function setMessage() {
    const { getAssemblyExports } = await globalThis.getDotnetRuntime(0);
    var exports = await getAssemblyExports("BlazorSample.dll");

    document.getElementById("result").innerText =
        exports.BlazorSample.JavaScriptInterop.Interop.GetMessageFromDotnet();
}

```

Make the `System.Runtime.InteropServices.JavaScript` namespace available at the top of the `Program.cs` file:

C#

```

using System.Runtime.InteropServices.JavaScript;

```

Load the module in `Program.cs` before `WebAssemblyHost.RunAsync` is called:

C#

```

if (OperatingSystem.IsBrowser())
{
    await JSHost.ImportAsync("Interop", "../js/interop.js");
}

```

`CallJavaScript2.razor`:

razor

```

@page "/call-javascript-2"
@rendermode InteractiveWebAssembly
@using BlazorSample.JavaScriptInterop

<h1>
    JS <code>[JSImport]</code>/<code>[JSEExport]</code> Interop
    (Call JS Example 2)
</h1>

@(message is not null ? message : string.Empty)

@code {
    private string? message;

    protected override void OnInitialized()
    {
        message = Interop.GetWelcomeMessage();
    }
}

```

CallDotNet2.razor:

razor

```

@page "/call-dotnet-2"
@rendermode InteractiveWebAssembly
@using BlazorSample.JavaScriptInterop

<h1>
    JS <code>[JSImport]</code>/<code>[JSEExport]</code> Interop
    (Call .NET Example 2)
</h1>

<p>
    <span id="result">.NET method not executed</span>
</p>

@code {
    protected override void OnAfterRender(bool firstRender)
    {
        if (firstRender)
        {
            Interop.SetWelcomeMessage();
        }
    }
}

```

 Important

In this section's example, JS interop is used to mutate a DOM element *purely for demonstration purposes* after the component is rendered in [OnAfterRender](#). Typically, you should only mutate the DOM with JS when the object doesn't interact with Blazor. The approach shown in this section is similar to cases where a third-party JS library is used in a Razor component, where the component interacts with the JS library via JS interop, the third-party JS library interacts with part of the DOM, and Blazor isn't involved directly with the DOM updates to that part of the DOM. For more information, see [ASP.NET Core Blazor JavaScript interoperability \(JS interop\)](#).

Additional resources

- [JavaScript `\[JSImport\]`/\[JSEExport\]` interop in .NET WebAssembly](#)
- [JavaScript `\[JSImport\]`/\[JSEExport\]` interop with a WebAssembly Browser App project](#)
- API documentation
 - [\[JSImport\] attribute](#)
 - [\[JSEExport\] attribute](#)
- In the `dotnet/runtime` GitHub repository:
 - [Configuring and hosting .NET WebAssembly applications](#) [↗](#)
 - [.NET WebAssembly runtime](#) [↗](#)
 - [dotnet.d.ts file \(.NET WebAssembly runtime configuration\)](#) [↗](#)